数据库复习

零、复习重点

```
// 感谢@KW
什么是数据库？
DBMS、DBS区别
数据库用户在做什么？
文件系统、数据库的区别
数据库结构，三级模式两级映像
剩下的了解即可

数据模型
三种模型都在讲什么
必考：需求->ER模型(主键要画、一对一一对多关系要画，实体属性鉴别规则)->关系表（模型） 参看ppt

关系模型
元组相关逻辑术语要搞清楚

关系代数、SQL语言 需要熟练度、常见套路如果现想可能比较难 练习题要着重复习
集中出聚集分组、相关子查询等题目，难点设置在这里
逻辑蕴含、全称量词 太难 一般不出 看一看即可

存储过程主要为触发器做铺垫，主要考触发器
系统编程、外部编程不会考察

不同索引的使用方式、数据结构与优势

完整性约束、六种约束方法要能判断清楚

安全性，
role的含义
grant，revoke，with grand option的用法

事务肯定有大题
ACID特性
事务调度的各种类型判断

备份、恢复方式（用什么日志）
缓冲区

关系理论
几部分综合起来来看
模式分解必有大题

最新一节课主要看概念即可
```

一、概述

1. 基础概念

（1）数据：用于记录信息的符号，可定性或者定量描述

信息：数据经过解释后表达出的意义（信息 = 数据 + 解释）

（2）数据管理的任务：数据存储、维护、查询、安全

（3）**数据库系统 DBS = 数据库 DB + 数据库管理系统 DBMS + 应用程序 + 用户 + 相关软硬件环境**

数据库DB：存储介质上的一个或一组文件，用来保存大量、有组织、可持久化的数据。不能直接编辑。

数据库管理系统DBMS：管理数据库的大型复杂软件系统

数据库系统DMS：完整的信息系统。

数据库用户：数据库管理员、应用系统开发人员（用DML使用数据）、终端用户

2. 发展历史

- 人工处理阶段：无OS，用磁带、卡片机，使用批处理，数据与程序高度绑定
- 文件系统阶段：使用文件形式，可以对数据存取管理。问题：
  - 数据冗余度大（各应用间数据冗余）、数据独立性差（依赖文件数据结构）、数据一致性差（同个数据多个文件，只更新部分）
- 数据库系统阶段：
  - 数据结构化：还描述数据之间的联系，结构化
  - 数据共享：不同用户程序可共享，低冗余
  - 数据独立性：物理独立性（修改物理存储方式，不必重写应用程序）、逻辑独立性（修改数据库的逻辑结构，不必重写应用程序）
  - 方便的用户接口：DDL、DML、DQL、DCL
  - 统一数据管理与控制：数据完整性、安全性、并发控制

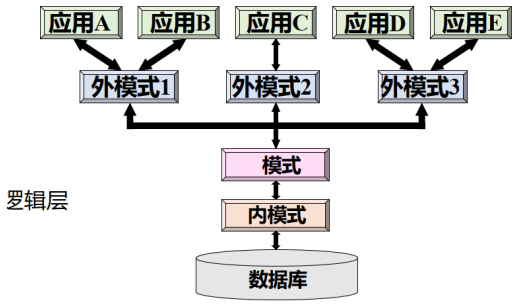
		人工管理阶段	文件系统阶段	数据库系统阶段
背 景	应用背景	科学计算	科学计算、管理	大规模数据管理
	硬件背景	无直接存取存储设备	磁盘、磁鼓	大容量磁盘、磁盘阵列
	软件背景	无操作系统	有文件系统	有数据库管理系统
	处理方式	批处理	联机实时处理， 批处理	联机实时处理，分布处理， 批处理
特 点	数据的管理者	用户(程序员)	文件系统	数据库管理系统
	数据面向的对象	某一程序	某一应用	平台系统(一个企业、跨国公司等)
	数据的共享程度	无共享，冗余度极大	共享性差，冗余度大	共享性高，冗余度小
	数据的独立性	不独立， 完全依赖于程序	独立性差	具有高度的物理独立性和一定 的逻辑独立性
	数据的结构化	无结构	记录内有结构，整体无结构	整体结构化，用数据模型描述
	数据控制能力	应用程序自己控制	应用程序自己控制	由DBMS提供数据安全性、完整性、并发控制、备份回复等能力

3. 数据库结构

（1）模式与实例

- 模式schema：数据库逻辑结构和特征的描述，反映数据的结构及其联系，即“表头”
  - eg. `Student(学号, 姓名, 性别, 年龄)`
- 实例Instance：模式在某一时刻的具体数据值，反映数据库某一时刻的状态，即一组数据

（2）三级模式结构：外模式、模式（概念模式）、内模式



其中：外模式多个，模式只有一个，内模式只有一个

（2.1）**模式**：逻辑层，概念模式

它是数据库中全体**数据的逻辑结构和特征**的描述，是所有用户的公共数据视图。

模式定义的内容包括：

1. 数据的逻辑结构：数据项名称、类型、取值范围等。
2. 数据之间的联系。
3. 数据相关的安全性、完整性要求等。

```
Student(SID, Name, Gender, Age)
Course(CID, CName, Credit)
Takes(SID, CID, Grade)
```

（2.2）**外模式**：用户层，子模式/用户模式

数据库用户使用的局部数据的逻辑结构和特征描述，通常基于视图 View 实现。

把模式里面的内容通过一些规则映射成一个新的视图，给用户层看，通常是模式的子集，一个数据库可以有多个外模式。

（2.3）**内模式**：物理层，存储模式

它描述数据的**物理结构和存储方式**，是数据在数据库内部的表示方式，即怎么存在硬盘上。比如记录如何存储：顺序存储、B 树结构、Hash 方法等；索引如何组织。

（3）二级映像

上面三个模式之间的映射转换关系。包括：

1. 外模式/模式映像
2. 模式/内模式映像

（3.1）**外模式/模式映像**

定义外模式和概念模式之间的对应关系，保证数据的**逻辑独立性**（逻辑层，即数据库结构变化时，只需要修改映像即可，用户层的调用还是不变，不用修改用户程序）

（3.2）**模式/内模式映像**

定义数据库全局逻辑结构与物理存储结构之间的对应关系。保证数据的**物理独立性**（如果数据库的存储结构改变了，DBA 只需要修改模式/内模式映像，让模式保持不变）。

（4）数据库功能结构

- 查询处理器：查询分析、检查、优化、执行
- 存储管理器：根据查询执行层的操作指令，在数据存储器中完成数据的增删改查
- 安全管理器：访问认证、权限控制、数据保护、审计机制、安全测试

二、数据模型—概念模型

1. 数据模型定义

（1）模型：对现实世界的抽象。

数据模型：是现实世界数据特征的抽象，是用来描述数据的一组概念和定义。

（2）三类数据模型：**概念数据模型、逻辑数据模型、物理数据模型**

- 概念数据模型：对现实世界的第一层抽象，提取事物对象的共同特征，对数据建模
    - 如：ER模型
- 逻辑数据模型：第二层，直接面向数据库逻辑结构，用于定义和操纵数据库数据
    - 区分不同类型数据库的依据，决定性能和应用范围
    - 常见：层次模型、网状模型、关系模型
- 物理数据模型：数据在存储介质上的组织结构和访问机制，结构、顺序、访问路径等

2. ER数据模型

一种概念数据模型，实体-联系模型（Entity-Relationship Data Model）。

三个基本元素：

- 实体 Entity
- 属性 Attribute
- 联系 Relationship / Relation

（1）常用两种抽象手段

- **分类**：定义某一类概念，把有相似特征的分到一起
- **聚集**：定义某一类型由哪些特征组成

（2）实体与实体集（表与对象）

实体：客观存在并且可以相互区分的客观事物或抽象事件，“东西”

实体集：同类实体的集合，也可以简称为“实体”

（3）属性

属性：实体所具有的某一特征（列名）

属性类型：包含哪几种属性。属性类型不同的实体属于不同实体集

- 分类1
    - 单值属性：每个特定实体在该属性上唯一，如年龄
    - 多值属性：同一属性可以有多个，如电话号码等
- 分类2
    - 简单属性：不可再分的属性
    - 复合属性：可以划分为更小的属性，如“地址 = 邮编 + 省市名 + 区名 + 街道”

域：属性的取值范围/集合

键：实体集中能够**唯一标识**实体的属性或属性组，如学号，每个实体这个属性都不同

- **复合键**：两个或多个属性组合起来才能唯一标识实体
- **超键**：任意能够唯一确定实体的一组属性
- **候选键：** 最小的超键，再去掉一个属性就不能标识
- **主键：** 候选键中任选一个
- 关系：主键 ⊆ 候选键 ⊆ 超键

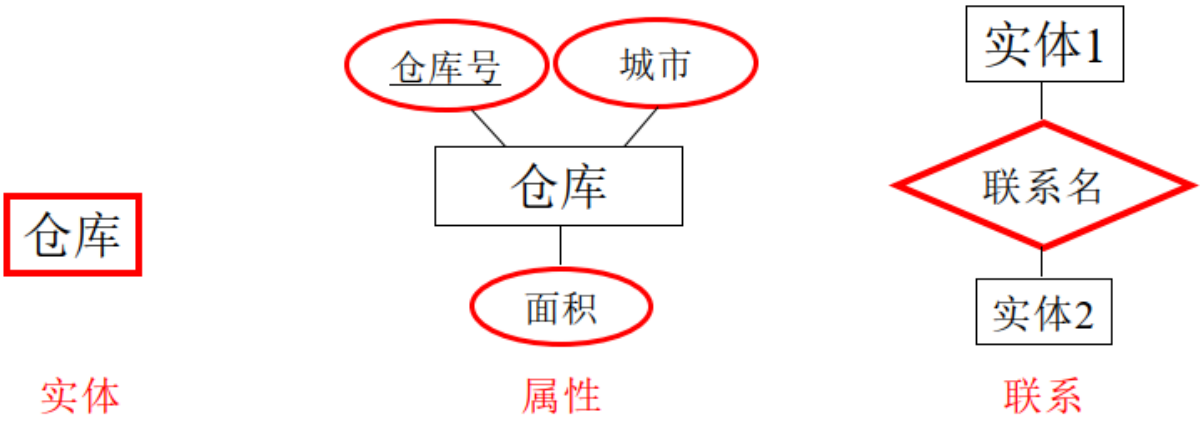
联系： 描述实体的对应关系，联系本质上是若干实体对的集合

- 联系可以看作一个新的表，其键是组成的复合键
- 除了两边的键，联系也可以有属性

3. ER图

表示：

符号	含义
矩形	实体
椭圆	属性
菱形	联系
连线	实体、属性、联系之间的关联
下划线属性	键属性
双椭圆	多值属性
复合椭圆结构	复合属性



（1）联系的元：参与联系的实体集个数

二元联系、三元联系、一元联系（排名、上下级、零件组成）

（2）联系的基数比约束：联系之间实体与实体关联的个数，可以标注在ER图上

- 一对一联系
- 一对多联系（班级与学生）
- 多对多联系（课程与学生）

（3）箭头的含义：表达由其他实体能唯一确定箭头指向的实体。比如Buy关系中，Store、Invoice、Person，就可以确定 Product。

（4）约束：对数据库的断言，数据库应该保证这个断言成立。

非空约束、唯一值约束、主键约束、外键约束、Check 约束

（5）ER建模步骤

1. 先设计实体类型：找出逻辑独立的事物，也就是管理对象。
2. 再设计联系类型：确定实体之间相互影响的关系。
3. 最后为实体和联系确定属性

4. ER建模原则

（1）忠于用户需求

（2）KISS准则：尽可能简单清晰

（3）避免冗余，不要用两个以上不同的实例描述同一件事情，且容易数据不一致

（4）能抽象为属性的，不要抽象为实体。实体判断标准：

- 这个事物有至少一个非键属性
- 这个事物处于 1:n 或 m:n 联系中“多”的那一端（一个属性不适合描述多个值）

三、数据模型—逻辑模型

1. 逻辑数据模型

定义：对现实世界的第二层抽象，负责**把概念模型映射成数据库的逻辑结构。**

三要素：数据结构 + 数据操作 + 数据约束

发展：文件模型（记录+字段）-> 层次模型（树形结构）-> 网状结构 -> 关系模型（二维表结构）

2. 关系模型概述

（1）定义：用二维表格结构来表示实体以及实体之间联系的数据模型，主要就看关系

（2）关系数据结构

表格说法	关系模型术语
表	关系 Relation
表名	关系名
列	属性 Attribute
行	元组 Tuple
单元格里的值	分量 Component
表头结构（框架）	关系模式 Relation Schema
表中具体数据	关系实例 Relation Instance

（3）部分集合论

（3.1） 域：一组具有相同数据类型的值的集合，也叫值域

- 域表示属性的取值范围。
- 域中值的个数叫**域的基数**（行数）。
- 域值没有排列次序

（3.2）笛卡尔积：一个集合（/二维表），是所有域中间元素排列组合的结果，用`x`表示。

笛卡尔积中的每一个元素叫**元组**，其中每一个量分别是这个元组的**分量**，元组有序。

基数即集合基数之积。

（3.3）关系：笛卡尔积的任意子集，且必须要有**现实意义。**

如果：

```
R ⊆ D1 × D2 × ... × Dn
```

那么 R 就是定义在这些域上的 n 元关系。

关系中属性的个数叫关系的**元数、目数或度**（列数）。

- 关系中不可以出现重复元素
- 属性（列）顺序原则上也无关
- 关系中不同属性可以来自一个列，但是必须有不同名字，名字称为**属性**
- 表中不能有表，属性值为原子的（不可再分），这种关系为规范化关系

3. ER模型与关系模型转化

（1）**实体**类型转换

规则：每个实体类型转换成一个关系模式；实体的属性变成关系模式的属性；实体标识符变成关系模式的键。

（2）**m:n联系**转换

m:n 联系通常转换成一个**独立的关系模式**（联系单独建表）。这个关系模式通常包含：

1. 两端实体的主键。
2. 联系自身的属性。
3. 主键通常是两端实体主键的组合。

（3）**1:n联系**转换

1:n 联系最好**合并到 n 端**实体对应的关系中，并添加外键

> eg. 系 —— 工作 —— 教师

转换时，把 1 端“系”的主键放到 n 端“教师”表中作为**外键。**

（4）**1:1联系**转换

1:1 联系可以合并到任意一端，但通常合并到更常用的一端。

也是合并添加为外键，合并到哪段看哪个表被查询更多。

（5）三元联系转换（简略）

三元联系类型	常见处理
1:1:1	可以与其中一个实体集合并
1:1:N	可以与 N 端实体集合并
1:M:N	通常转成独立关系模式
M:N:P	转成独立关系模式，主键通常由三个实体集主键组合

4. 关系完整性

定义：关系模型的完整性规则是对关系的一种约束条件，即数据应该满足的规则。

（1）域完整性：属性值必须符合该属性的取值范围（包括属性值是否可以为NULL）

（2）实体完整性：主属性（主键）不能为空，主属性取值必须唯一。

（3）参照完整性：来保证表与表之间的引用是合理的

- 外键定义

    - 设 F 是关系 R 中的一个或一组属性，但不是 R 的键。但如果 F 与关系 S 的主键 K 相对应，那么 F 是关系 R 的**外键。**
    - 外键只能两种取值：NULL 或者 等于被参照关系中某个元组的主键值
- 性质：
    - 关系 R 和 S 不一定是不同关系。（可能是自引用）
    - 外键与主键不一定同名

（4）用户定义完整性

四、数据模型—关系代数

1. 关系代数定义

关系代数是对关系的查询语言，用对关系的运算来表达查询。

- 运算对象是关系，运算结果是关系

四类关系运算符：

运算符		含义	运算符		含义
集　合 运算符	∪ ∩ － ×	并 交 差 广义笛卡尔积	比　较 运算符	＞ ≥ ＜ ≤ ＝ ≠	大于 大于等于 小于 小于等于 等于 不等于
专门的 关　系 运算符	σ	选择	逻　辑 运算符	¬	非
	Π	投影		∧	与
	⋈	连接		∨	或
	÷	除			

基本运算： 并、差 、广义笛卡尔积、选择、投影

- 其他运算可以由基本运算推导出来

2. 集合运算：并、交、差

主要对于行（元组）来说的。

（1）**相容条件**

两个关系R、S运算必须满足：

- R 和 S 的属性个数相同（列数一样）
- R 的第 i 个属性和 S 的第 i 个属性来自同一个域（属性名可以不同）

（2）并运算：`R ∪ S`

属于 R，或者属于 S 的所有元组组成的新关系

（3）差运算：`R − S`

属于 R，但不属于 S 的元组组成的新关系。

（4）交运算：`R ∩ S`

同时属于 R 和 S 的元组组成的新关系。推导关系（R减去[在R中但是不在S中的部分]）

```
R ∩ S = R − (R − S)
```

3. 广义笛卡尔积

也是集合运算，但是没上面的相容条件要求。

`R × S`：把 R 中每一行和 S 中每一行都拼接一次，得到所有可能组合。

度为两个关系度之和，元组个数为两个关系元组个数乘积。

4. 关系运算

选择、投影、连接、除

（1）选择：从关系中选出满足条件的元组（行）

$$\sigma_F(R) = \{t \mid t \in R \wedge F(t) = true\}$$

- 其中R是关系名，σ是选择运算符，F是逻辑表达式。

约等于WHERE中的条件。

（2）投影：从关系中选出指定的属性列，形成新的关系（可以选定集合，可以改变列顺序）

$$\pi_A(R) = \{r.A \mid r \in R\}$$

✓ 其中R是关系名，π是投影运算符，A是被投影的属性或属性集

- 也可能出现重复行，需要去重
- π也可以写成 `Π`

例如：

```
π_Name, Salary(Employee)
```

（3）连接：两个表之间的运算，两个表之间通常一对多联系

（3.1）θ连接

先笛卡尔积，再选出符合给定属性关系的元组。

从两个关系的广义笛卡儿积中选取给定属性间满足θ操作的元组。

$$R \underset{A\,\theta\,B}{\bowtie} S = \{ t \mid t = < r, s > \wedge r \in R \wedge s \in S \wedge r[A]\ \theta\ s[B] \}$$

θ为算术比较符，当θ为等号时称为等值连接；θ为“<”时，称为小于连接；θ为“>”时，称为大于连接等等。

这种叫做**Theta连接。**

（3.2）等值连接

即上述连接条件是等号时，为等值连接。但是会保留两边重复的属性列

（3.3）自然连接

自然连接是从两个关系的广义笛卡尔积中选取在**所有相同属性列上取值相等**的元组，并**去掉重复的属性列。**自然连接是特殊的等值连接。

$$R \bowtie S = \{ t \mid t = < r, s > \wedge r \in R \wedge s \in S \wedge r[B]=s[B] \}$$

自然连接 = 等值连接 + 去掉重复属性列。

（3.4）外连接

原问题：自然连接的时候，有的元组在部分表中可能没有表项，连接之后就直接丢失了。

外连接是给参与连接的表中附加一个取值全为空的行，和参与连接的另一方任何一个未匹配的元组都能匹配。

哪边加两个杠代表哪边信息全保留。

外连接 = 自然连接 + 失配的元组。

外连接的形式：左外连接、右外连接、全外连接 。

$R \supset\!\bowtie S$ 左外连接 = 自然连接 + 左侧表中失配的元组。

$R \bowtie\!\subset S$ 右外连接 = 自然连接 + 右侧表中失配的元组。

$R \supset\!\bowtie\!\subset S$ 全外连接 = 自然连接 + 两侧表中失配的元组。

（4）除

含义：满足全部条件的对象。至少包含 S 中所有项目的那些 X。

假设有：

```
R(X, Y)
S(Y)
```

那么：

```
R ÷ S
```

结果是：

```
所有这样的 X：对于 S 中的每一个 Y，(X, Y) 都在 R 中出现。
```

用数学语言写：

> 设有关系$R(X,Y)$和$S(Y)$，其中X、Y可以是单个属性或属性集，则：
>
> $$R \div S = \{r.X \mid r \in R \wedge Y_x \supseteq S\}$$

5. 其他

（1）关系代数能力限制：不能计算传递闭包

（2）解题方法总结

1️⃣ 需要连接的表可以先连接上

2️⃣ 根据要求完成筛选

3️⃣ 最后进行投影，看输出需要什么

```
π 最终要的属性 (
    σ 筛选条件 (
        需要的表连接起来
    )
)
```

GPT总结方法：

普通查询：先把需要的表连接起来 → 用 `σ` 按条件筛选行 → 最后用 `π` 投影题目要的列。

涉及多表：能自然连接就 `⋈`，不确定就写成 `× + σ连接条件`，最稳。

“没有 / 未 / 不在”：用差集，`所有对象 − 满足条件的对象`。

“既……又……”：用交集 `∩`，或分别筛选后相交。

“所有 / 全部 / 每一个”：用除法 `÷`。

“至少一个”：普通连接筛选即可。

“至少两个不同”：同一关系复制两份自连接，条件写“同一个对象 + 不同属性值”。

“最多一个 / 只一个”：转成 `总体 − 至少两个不同的`。

“三者同城 / 不同城 / 至少两者同城”：直接对城市属性写 `=`、`≠`、`∨` 条件。

五、SQL语言

（一）基本介绍

SQL = Structured Query Language（结构化查询语言）

特点：高度统一、高度非过程化、面向集合的操纵方式、两种使用方法（命令、程序）

组成部分：

类型	英文	作用	常见命令
查询语言 DQL	Query Language	查询数据	`SELECT`
定义语言 DDL	Data Definition Language	定义表、视图、索引等结构	`CREATE`、`DROP`、`ALTER`
操纵语言 DML	Data Manipulation Language	插入、删除、修改数据	`INSERT`、`DELETE`、`UPDATE`
控制语言 DCL	Data Control Language	授权、完整性规则、事务控制等	`GRANT`、`REVOKE`

（二）查询语言：单表查询

常见结构：

```
SELECT ... 选定显示的属性列
FROM ... 指定查询对象（来自于谁），可以是基本表或视图
WHERE ... 指定元组筛选条件
GROUP BY ... 按指定列分组
HAVING ... 对分组后的组进行筛选
ORDER BY ...; 对最终结果排序
```

先讲**单表查询**：

执行顺序：**FROM->WHERE->GROUP BY->HAVING->SELECT->ORDER BY**

1. SELECT

SELECT 表示最后结果要显示的列以及顺序。

（1）可以填在后面的变量（变量之间加逗号）

- 直接填列名（属性名）
- * 表示全部
- 算数表达式 2026-Age
- 字符串常量 “text”
- 函数 LOWER(Name)
- 列别名（使用AS，可省略，直接加空格） 2026-Age **AS** Birthday

（2）是否行可以重复

- **ALL** 关键字：保留所有重复行，默认可以不加
- **DISTINCT** 关键字：消除重复值的行

这个关键字的范围是**整个目标列**，不能单独作用于某一列

2. WHERE

WHERE子句表示查询条件。

（1）比较大小：`WHERE Age < 20`

```
=    等于
<>   不等于
!=   不等于
>    大于
<    小于
>=   大于等于
<=   小于等于
```

（2）确定范围：谓词 **（NOT）BETWEEN … AND …**

包含边界，可以填数字

（3）确定集合：谓词 **（NOT）IN <值表>**

值表为一组取值，相当于多个OR的简写。

```
WHERE Sdept IN ('IS', 'MA', 'CS')
```

（4）字符串匹配：**LIKE / NOT LIKE <模式>**

模式是一个字符串/字符串模版，不加通配符就相当于完整字符串相等匹配（类似于=）。通配符：

通配符	含义
`%`	任意长度字符串，长度可以为 0
`_`	任意单个字符

- 如果本身要查询的字符串有%或者_（通配符），那么需要用**ESCAPE**标明转义符，比如：

  ```
  WHERE Cname LIKE 'DB\_Design'
  ESCAPE '\';
  ```

  _就表示普通下划线，MySQL不能用\做转义符。
- 若包含单引号的字符串，表达式用双引号代替单引号

（5）多重匹配：使用逻辑表达式 **AND、OR、NOT**

（6）WHERE语句涉及空值（NULL）的时候：

- 条件表达式（WHERE里面）只有三种结果：TRUE, FALSE, UNKNOWN

- 任何值与空值NULL比较都会返回**UNKNOWN**
- WHERE最终**只选择结果为TRUE**的记录

测试是否为空值不能用等于，需要用：**IS NULL / IS NOT NULL**

- 三值逻辑运算：TRUE=1，FALSE=0，UNKNOWN=1/2，And=Min，Or=Max，Not(x)=1-x

3. ORDER BY

按照一个或多个属性排序：

```
ORDER BY 属性 [ASC|DESC]
ORDER BY Sdept, Sage DESC;
```

其中后面关键字（跟属性对应）表示升序/降序，ASC升序（默认，小到大），DESC降序

空值NULL当做正无穷用，ASC放最后，DESC放最前

- ORDER必须出现在SELECT语句最后部分

4. 集函数

放到SELECT结果列里面

函数	含义
`COUNT`	计数
`SUM`	求和
`AVG`	求平均
`MAX`	求最大值
`MIN`	求最小值

```
COUNT([DISTINCT|ALL] 列名)
```

DISTINCT/ALL表示统计时算不算重复行，COUNT中可以直接（*），表示全部

- 空值不会加入计算

5. GROUP BY

用于**分组**，常**和集函数一起用**：使集函数作用于每个组，而不是作用于整个查询结果（那样最后就一行了）

```
GROUP BY 属性/集
```

- 分组时值相等并为一组，每个组只产生一行结果
- SELECT子句列名中只能包含分组属性（GROUP BY后面的属性）和集函数，不能用里面成员单独的值

6. HAVING

用于筛选分组后的组（WHERE作用于分组之前，所以这里相当于WHERE的作用）

WHERE不可以使用集函数，0而HAVING可以。

（三）查询语言：多表查询

1. 多表查询定义

`FROM` 子句中涉及多个表的查询。

主要形式：`表名1.列名1 比较运算符 表名2.列名2`

基本套路：

```
SELECT 要显示的列
FROM 表1, 表2, 表3
WHERE 表1.连接字段 = 表2.连接字段
  AND 表2.连接字段 = 表3.连接字段
  AND 其他筛选条件;
```

2. 广义笛卡尔积

不带任何谓词（连接条件），全部排列组合。

```
SELECT Student.*, SC.*
FROM Student, SC;
```

SQL 92也可以使用**CROSS JOIN（交叉连接）**，等价：

```
SELECT *
FROM Student CROSS JOIN SC;
```

3. 等值连接

Where中**运算符为等号**的连接。

若两个表比较的属性有同名属性，则必须加表前缀。

4. 自然连接

去掉所有重复属性列。它会自动按所有同名属性相等连接。

5. 非等值连接

连接运算符不是等号，而是：`> < >= <= != BETWEEN`

SQL 92中连接操作还可以使用**内连接：(INNER) JOIN ... ON ...**（inner可省略）

```
SELECT 属性列表
FROM 表1 [INNER] JOIN 表2
ON 连接条件
WHERE 其他限制条件;
```

6. 自连接

一个表和自己连接，必须要起别名（FROM中加AS），WHERE必须使用别名前缀。

比如查每门课的间接选修课：

```
SELECT FIRST.Cno, SECOND.Cpno
FROM Course AS FIRST, Course AS SECOND
WHERE FIRST.Cpno = SECOND.Cno;
```

7. 外连接

- 内连接：只保留匹配上的
- 外连接：匹配上的保留，没匹配上的也保留，用 NULL 补另一边

类型	含义
`LEFT JOIN`	保留左表全部元组(左边行没匹配上的，补NULL保留)
`RIGHT JOIN`	保留右表全部元组
`FULL JOIN`	左右两表未匹配元组都保留

```
SELECT 属性列表
FROM 表1 LEFT OUTER JOIN 表2
ON 连接条件;
```

必须用ON表示连接条件。

8. 复合条件连接

WHERE子句有多个连接条件。

（四）子查询

定义：把一个查询块嵌套在另一个查询块的 `WHERE` 子句或 `HAVING` 短语的条件中，称为**嵌套查询**，也就是**子查询**。

- 子查询中不能使用ORDER BY，必须放在最外层
- 子查询还可以插入到From语句，必须设置别名。
- 也可以插入到Select语句，但必须是单行单列。

两种分类：

- 不相关子查询：子查询不依赖外层查询，可以先独立执行，然后把结果交给外层查询
- 相关子查询：子查询的条件依赖外层查询当前元组的值，不能独立求解，外层查询读取一行，需要把这一行结果传到子查询里面用。

1. IN子查询

```
WHERE 属性 IN (子查询)
```

表示这个属性值属于子查询结果集合。

比如查询与刘晨在同一个系学习的学生：

```
SELECT Sno, Sname, Sdept
FROM Student
WHERE Sdept IN
      (SELECT Sdept
       FROM Student
       WHERE Sname = '刘晨');
```

子查询和连接查询可以等价

2. 比较运算符子查询

如果能确定子查询**只返回一个值**，可以用比较运算符，比如例如，假设一个学生只属于一个系，且名字唯一：

```
SELECT Sno, Sname, Sdept
FROM Student
WHERE Sdept =
      (SELECT Sdept
       FROM Student
       WHERE Sname = '刘晨');
```

子查询必须跟在比较符后面！

3. 带ANY与ALL的子查询

- ANY：任意一个值
- ALL：所有值

通常和比较运算符一起用。

比如：

```
Sage < ANY(...子查询)
```

Sage只要小于任何一个值即可。

4. EXISTS子查询

一般是相关子查询（需要用到外部查询的值）

表示是否存在，即子查询的表是不是空的，只返回TRUE/FALSE，相当于存在量词“∃”

例如：查询选修了 1 号课程的学生姓名

```sql
SELECT Sname
FROM Student
WHERE EXISTS
      (SELECT *
       FROM SC
       WHERE SC.Sno = Student.Sno
         AND SC.Cno = '1');
```

所有带 `IN` 谓词、比较运算符、`ANY` 和 `ALL` 谓词的子查询，都能用带 `EXISTS` 谓词的子查询等价替换；但有些 `EXISTS / NOT EXISTS` 查询不能被其他形式等价替换。

5. NOT EXISTS

> 看看就行？似乎太难了不考？

（1）全称量词

不存在就是相当于 `¬∃`。SQL 语言中没有全称量词 `For all`，可以把全称量词转换成存在量词：`∀x P ≡ ¬∃x(¬P)`。

```
“对所有课程都选了”
=
“不存在一门课程没选”
```

如：查询选修了全部课程的学生姓名

```sql
SELECT Sname
FROM Student
WHERE NOT EXISTS
      (SELECT *
       FROM Course
       WHERE NOT EXISTS
             (SELECT *
              FROM SC
              WHERE SC.Sno = Student.Sno
                AND SC.Cno = Course.Cno));
```

（2）逻辑蕴含

用谓词运算表达逻辑蕴含：

```
p → q ≡ ¬p ∨ q
∀y(p → q) ≡ 不存在 y，使得 p 成立且 q 不成立
```

如：查询至少选修了学生 95002 选修的全部课程的学生号码

= 不存在这样一门课：95002 选了这门课，而学生 x 没选这门课

```
SELECT DISTINCT SCX.Sno
FROM SC AS SCX
WHERE NOT EXISTS
      (SELECT *
       FROM SC AS SCY
       WHERE SCY.Sno = '95002'
         AND NOT EXISTS
             (SELECT *
              FROM SC AS SCZ
              WHERE SCZ.Sno = SCX.Sno
                AND SCZ.Cno = SCY.Cno));
```

```
SCY：95002 选过的课程
SCX：候选学生 x
SCZ：检查 x 是否选了 SCY 中这门课程
```

（五）集合查询

SQL	关系代数	含义
`UNION`	并	合并两个查询结果
`INTERSECT`	交	取两个查询结果共有部分
`EXCEPT`	差	从前一个结果中去掉后一个结果

集合操作两边必须**相容**：

- 列数相同
- 对应列类型兼容

比如：

```
SELECT Sno FROM Student
UNION
SELECT Sno FROM SC;
```

（六）数据定义语言：DDL

数据库模式 `schema` 主要由数据库中关系的声明构成，也包括视图、索引、触发器等对象；这些对象的定义通过 SQL 的 DDL 部分完成。

三种：CREATE、DROP、ALTER

1. 定义基本表 CREATE

```
CREATE TABLE <表名>
(
  <列名> <数据类型> [<列级完整性约束条件>],
  ...
  [<表级完整性约束条件>]
);
```

2. 数据类型

数据类型	含义
`INT` / `INTEGER`	整数
`REAL` / `FLOAT`	实数 / 浮点数
`CHAR(n)`	定长字符串
`VARCHAR(n)`	变长字符串，最大长度为 n
`DATE`	日期
`TIME`	时间
`DATETIME`	日期时间

3. 完整性约束

约束	含义
`PRIMARY KEY`	主键约束（取值唯一，只能有一个主键(可复合)，非空），复合时使用表级约束
`UNIQUE`	唯一性约束（某列取值唯一，不一定非空）
`NOT NULL`	非空约束
`REFERENCES`	参照完整性约束，外键引用，必须引用已存在的外键

4. 删除表 DROP

```
DROP TABLE 表名;
```

5. 修改表 ALTER

```
ALTER TABLE 表名
    [ADD 新列名 数据类型 [完整性约束]]
    [DROP 列名/完整性约束名]
    [MODIFY 列名 数据类型];
```

- ADD：增加新列，不论基本表中原来是否已有数据，新增加的列一律为空值。

- DROP：删除列或者约束，如`ALTER TABLE Student DROP UNIQUE(Sname);`
- MODIFY：修改列定义，可能破坏数据，`ALTER TABLE Student MODIFY Sage SMALLINT;`

（七）数据更新语言：DML

改变数据库内容；数据更新包括`INSERT`、`DELETE`、`UPDATE`三种。

1. INSERT INTO：插入数据

（1）插入单条元组

```
INSERT
INTO 表名 [(属性列1, 属性列2, ...)]
VALUES (常量1, 常量2, ...);
```

其中属性列用于表示插入的属性列的顺序，没指定的属性取空值。比如：

```
INSERT INTO Student(Sno, Sname, Ssex, Sage, Sdept)
VALUES ('95020', '陈冬', '男', 18, 'IS');
```

（2）插入子查询结果

```
INSERT INTO 表名 [(属性列1, 属性列2, ...)]
SELECT ...
FROM ...
WHERE ...;
```

也要保证值的个数与类型相等。

2. UPDATE SET：修改数据

修改表中满足WHERE条件的元组，不带WHERE则修改所有元组。

```
UPDATE 表名
SET 列名 = 表达式,
    列名 = 表达式,
    ...
[WHERE 条件];
```

WHERE可以加子查询：将计算机系全体学生成绩置为0

```
UPDATE SC
SET Grade = 0
WHERE 'CS' =
      (SELECT Sdept
       FROM Student
       WHERE Student.Sno = SC.Sno);
```

3. DELETE：删除数据

```
DELETE FROM 表名
[WHERE 条件];
```

也可以加子查询。

（八）视图：View

定义：视图是一种虚表，是从一个或几个基本表或视图导出的表；视图其实是在数据字典中存储的一条 `SELECT` 语句。

1. 定义视图

```
CREATE VIEW 视图名 [(列名1, 列名2, ...)]
AS 子查询
[WITH CHECK OPTION];
```

- 列名可以忽略，如果有表达式或者聚集函数建议写列名。
- 不建议使用SELECT *创建视图，因为表结构可能变化，可扩展性差。

WITH CHECK OPTION：更新视图的时候不能破坏视图中定义的条件，即新的更新放进视图走一圈才能满足要求。

2. 视图查询

与基本表查询完全一样

3. 视图更新

与基本表更新基本相同。

部分视图无法更新，因为更新视图会被转换为基本表更新，但是有些集函数等无法转换。

4. 视图删除

DROP VIEW 名字。

不会删除基本表里面的数据。

5. 视图的作用

（1）简化查询

（2）提供安全保护，只给用户提供视图

（3）保证逻辑独立性：表结构发生变化时，只需要改变视图的映射，用户层程序不用改

（4）限制更新范围：WITH CHECK OPTION

六、数据库编程

对数据的分析处理主要由数据库应用程序来完成。两种形式：

- 数据库内部应用程序：执行效率高，用于例行分析维护

    - 存储过程：CALL调用（完整过程），返不返回均可 PRECEDURE
    - 函数：SQL表达式调用，必须有返回值 FUNCTION
    - 触发器：表删改时自动触发，不返回 TRIGGER
- 数据库外部应用程序：开发灵活，功能和可移植性强

（一）存储过程

把一段经常重复执行的 SQL 逻辑封装起来。**用过程化 SQL 写好的一段程序，编译优化后保存在数据库服务器里，使用时直接调用。**

优点：

- 模块化：复杂业务封装为一个模块
- 效率较高：创建时候预编译、优化，每次直接调用
- 减少网络传输：程序在数据库服务器端执行，不需要外部程序反复传大量数据
- 安全性较好：参数化调用可以减少 SQL 注入风险。

（1）基本形式

定义：

```
CREATE PROCEDURE procedure_name (
    [ IN | OUT | INOUT ] param_name type [, ...]
)
[ BEGIN ]
    sql_statement
[ END ]
```

- `procedure_name`：存储过程名字。
- `param_name type`：参数名和参数类型
- `IN | OUT | INOUT`：参数方向。
- `sql_statement`：过程体，也就是内部执行的 SQL 和过程控制语句。

用INTO表示传出的参数，例如：

```
CREATE PROCEDURE proc_name(IN p_id INT, OUT p_count INT)
BEGIN
    SELECT COUNT(*) INTO p_count
    FROM student
    WHERE class_id = p_id;
END;
```

调用：

```
CALL procedure_name(实参列表);
```

删除：

```
DROP PROCEDURE procedure_name;
```

（二）过程化SQL

变量、嵌入式 SQL、动态 SQL、游标。

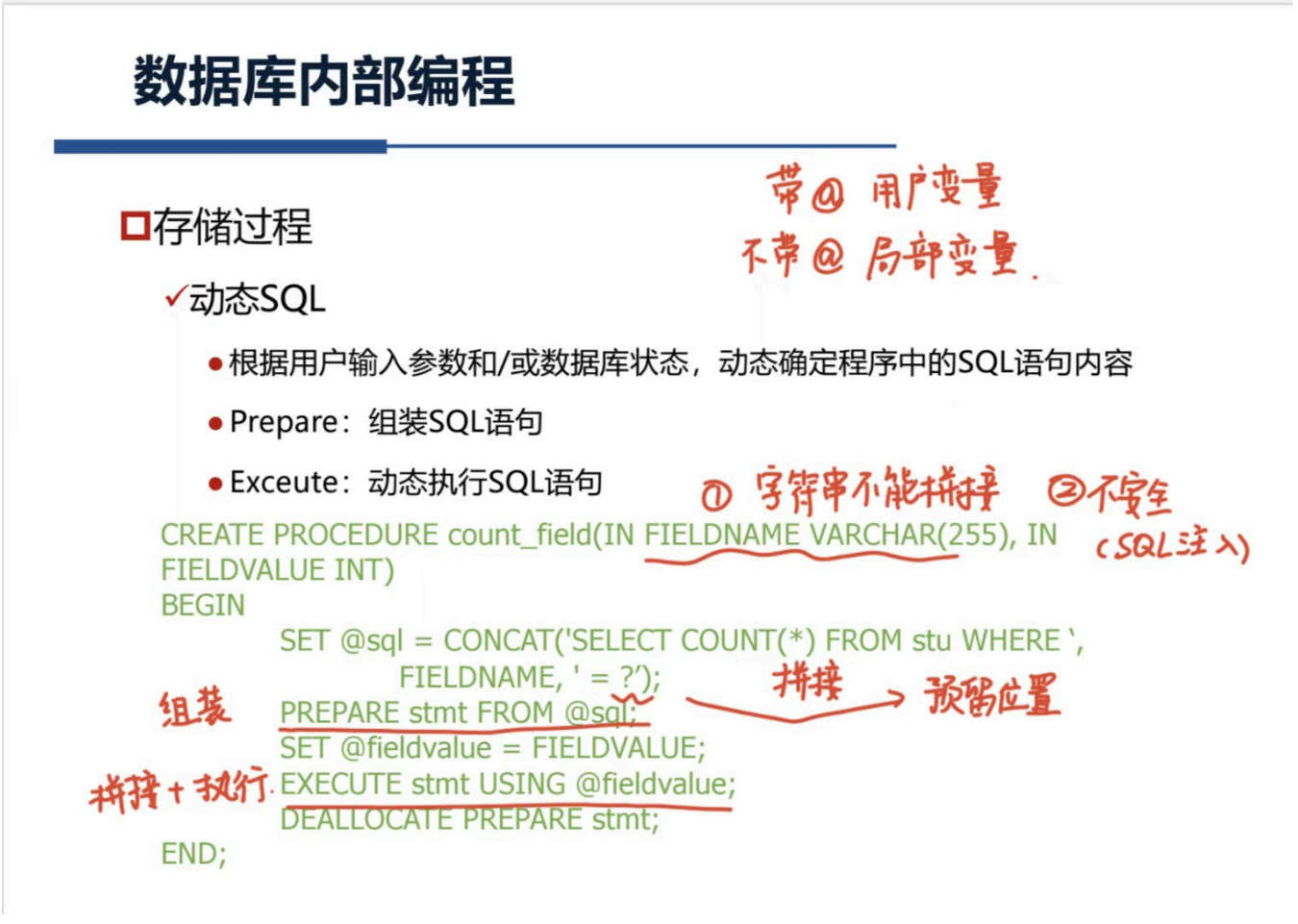
1. 变量

声明变量**DECLARE**，然后从字段中读取赋值，然后修改变量，之后把变量再使用**SET**写回表。

```
DECLARE 变量;
SELECT 字段 INTO 变量 FROM 表 WHERE 条件;
SET 变量 = 表达式;（比如++）
UPDATE 表 SET 字段 = 变量 WHERE 条件;
```

2. 动态SQL

动态 SQL 是运行时根据参数拼出 SQL。



`ONCAT`：拼接 SQL 字符串。

`PREPARE`：把字符串形式的 SQL 准备成可执行语句。

`EXECUTE`：执行准备好的 SQL。

`USING`：给 SQL 中的 `?` 占位符传值。

`DEALLOCATE PREPARE`：释放预处理语句。

3. 游标编程

- 正常SELECT -> 返回集合
- 游标 -> 每次只处理一部分（比如**逐行**）

主要步骤：

```
DECLARE   声明游标
OPEN      打开游标
FETCH     取当前/下一行
CLOSE     关闭游标
```

```
DECLARE cursor_name CURSOR FOR
    SELECT ...;

OPEN cursor_name;

FETCH cursor_name INTO 变量;

CLOSE cursor_name;
```

示例，统计 `t_user` 表中年龄小于 `c_age` 的人数：

游标编程示例

```
CREATE PROCEDURE stu_count(c_age int)
BEGIN
    DECLARE p_age int; # 声明变量
    DECLARE p_c int;
    -- 声明游标结束判断变量，默认值为0;
    DECLARE fetchSeqOk boolean DEFAULT 0;
    DECLARE my_cursor CURSOR for select age FROM t_user; -- 定义游标
    -- 游标执行结束时将会设置fetchSeqOk 变量为1
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET fetchSeqOk = 1;
    -- 在MySql中，造成游标溢出时会引发mysql预定义的NOT FOUND错误
    SET p_c=0;
    # 打开游标
    OPEN my_cursor;
        WHILE fetchSeqOk=0 DO -- 判断是不是到了最后一条数据
            fetch my_cursor into p_age; -- 游标改变位置指向下一行，取下一行数据
            IF p_age<c_age THEN
              SET p_c=p_c+1;
            END IF;
        END WHILE;
        Select p_c as concat('小于', c_age, '岁的总人数'); -- 输出结果
    CLOSE my_cursor;  -- 关闭游标，释放内存
END
```

（三）函数编程

放在SELECT后面参数里面。

```
CREATE FUNCTION capitalize(input_string VARCHAR(255))
RETURNS VARCHAR(255)
BEGIN
    DECLARE output_string VARCHAR(255);

    SET output_string =
        CONCAT(
            UPPER(LEFT(input_string, 1)),
            LOWER(SUBSTRING(input_string, 2))
        );

    RETURN output_string;
END;

---
使用：
SELECT capitalize('samPLeSTring');

SELECT capitalize(stu_name)
FROM stu;
```

对比：

对比项	存储过程	函数
创建	`CREATE PROCEDURE`	`CREATE FUNCTION`
调用	`CALL proc(...)`	`SELECT func(...)`
返回值	可无返回值，可用 OUT 参数	必须 `RETURNS` + `RETURN`
适用	完成一组业务操作	计算并返回一个值
能否嵌入表达式	一般不行	可以

（四）数据库外部编程

外部访问流程：

```
建立连接
创建游标/语句对象
执行 SQL
获取结果
关闭资源
```

七、索引&查询优化

（一）计组/OS知识

（1）物理存储分级

1级内存->2级磁盘->3级大容量存储器

内存放常用数据、缓存；磁盘放业务数据；三级放归档数据。

非易失性存储器

（2）磁盘结构

柱面-盘面-扇区

扇区通常 512 字节；**页面/块**由多个连续扇区组成，比如 4KB、8KB。

磁盘读写时延：寻道 + 旋转 + 读写

数据库优化：多利用顺序IO

（3）数据存储结构

- 块 ：磁盘页面，最小分配单元
- 区：连续块的组合，盘区
- 段：多个区的组合，类似于分段式程序，一个段存储一类数据
- 表空间：
    - 逻辑上：表空间由 0 个或多个段组成。

      物理上：表空间由一个或多个数据文件组成。

      一个段不能跨越一个表空间，但可以跨越表空间内的多个文件。

      表空间的作用：把逻辑结构和物理结构统一起来。
- 数据库：一个或多个表空间

（4）文件组织结构

数据文件中页面之间、记录之间如何组织。

- 堆组织表
- 索引组织表
- 聚簇表，包括索引聚簇表、散列聚簇表。

堆组织表： 记录顺序没有限制，简单放进文件里，数据排列顺序不可预测。

插入简单，但是没有索引只能顺序遍历查找。

（二）索引

索引是主表上的一种辅助数据结构，对表中**一个或多个列**的值进行排序，用于提高查询速度。

索引一般包括：键值 + 指针（指向记录）

1. 单级索引

单级索引可以二分查找，但是磁盘IO高，海量数据性能也不高。

2. 多级索引：B+树

多叉树，百万请求只要3次IO，范围查找很快。

（1）特点：

- 是平衡树。
- 从根节点到所有叶节点路径长度相同。
- 非叶节点只存索引键和指针。
- 真正的数据指针/RID 存在叶节点。
- **叶节点之间用链表连接。**
- 每个节点一般对应一个磁盘页。
- 查询复杂度是 $O(\log_m n)$，m 是分叉数，也就是 B+树的阶。

（2）查找过程：

- 从根节点开始。
- 根据索引键（大于、小于分边）判断该往哪个子树走。
- 到中间节点继续判断。
- 最后到叶子节点。
- 在叶子节点找到 key 和对应 RID。
- 根据 RID 去主表取记录。

（3）与B树对比：

B树是每个节点（包括非叶）都存数据，每个节点能放的数据变少，树变高。

同时B+树叶节点间有链表，范围查询效率更高（不用每个值都从根节点查），缓存命中也高。

B+树更新维护代价小。

（4）插入过程

如果位置不够，就一直向上分裂，直到根节点。

- 先找到应该插入的叶节点。
- 如果叶节点有空位，直接插入。
- 如果叶节点满了，分裂成两个叶节点。
- 分裂后，把新的分隔键插入父节点。
- 如果父节点也满，父节点继续分裂。
- 如果根节点满并分裂，就新建一个根节点，树高加 1。
- 插入后 B+树仍然保持平衡。

（5）应对重复键值

只建一个路径，然后用链表存储重复键值对应记录行的 RID。

（6）劣势

只有在查询的记录数占总数比例特别高（比如百分之60），不用索引可能更快，不如全表扫一遍。

3. 散列Hash索引

一般只需要一次磁盘IO。

把记录分散存到多个“桶”中，一个桶是一个数据块，里面存多个数据，存取按照整个桶存取。

1. 给定索引值 V。
2. 通过 hash 函数算出桶编号 K。
3. 读取编号 K 的桶。
4. 在桶内顺序查找目标记录/RID。

如果一个桶溢出，需要建立溢出块，挂到桶的链上。

（1）优点

CPU密集型，IO少

等值查找很快（查对应的数据）

（2）缺点

不支持范围查找（因为不是按照值索引，每个值都得重复查找）

没有顺序，不适合order by、group by

不适合重复值很多的列（哈希冲突，增加IO）

重构代价大

4. 聚簇索引 Cluster

规定数据在**物理文件中**的存储顺序，数据表中记录的物理存放顺序和索引键的逻辑顺序一致。

一张表**只能有一个聚簇索引**，因为物理顺序只能用一个

优点：

- 查询速度快于B+树
- **范围查找能力强**：数据在物理页上按顺序排列，重复值也排在一起。所以范围查询、`between`、`<`、`<=`、`>`、`>=`、`group by`、`order by` 可能明显变快

缺点：

- 一个表只能有一个聚簇索引，不能按照多种列建多种索引
- 数据维护开销大，不适合DML频繁的表

5. 联合索引

多个字段一起组成索引键：

先按 a 排；a 相同再按 b 排；a、b 都相同再按 c 排。

最左前缀原则：必须在索引中使用最左边的索引前缀（a），索引才能生效

6. 索引失效

```
select * from Emp where age / 2 > 20;
```

不要在索引列上做计算、函数、转换，直接用他比较或者范围比较。

如果实在不行，做**单独的函数索引。**

（三）查询优化

> 同一个 SQL 可以有很多执行方案，不同执行方案代价差距巨大。

1 几种不同方案：

（1）先做笛卡尔积，再选择

（2）先做连接，再选择

（3）先做选择，再连接

（4）有索引

2 一般准则：

- 选择运算尽可能先做，减少中间关系
- 连接前先做预处理（连接属性的排序、索引）
- 投影和选择同时做
- 投影和双目运算结合
- 提取公共子表达式

3 一般过程

- 将 SQL 转换成内部表示，通常是语法树。
- 根据等价变换规则把语法树转换成优化形式，也叫代数优化。
- 选择底层操作算法，也叫物理优化。
- 生成查询计划，选择代价最小的执行方案。

八、完整性约束

（一）定义

（1）数据库完整性：数据的正确性和相容性

（2）完整性控制机制：

- **完整性约束条件定义机制**：DBMS 要允许你定义规则，例如 `PRIMARY KEY`、`CHECK`、`FOREIGN KEY`、触发器等
- **完整性检查机制**：用户执行 `INSERT / UPDATE / DELETE` 时，DBMS 检查这次操作会不会违反约束
- **违约反应**：如果违反约束，DBMS 要采取动作，比如拒绝执行、级联删除、置空、触发器报错等

（3）完整性约束对象维度：

- 列级：只对一个属性约束
- 元组级：同一行多个属性之间的约束
- 关系级：多行、多表

（4）状态分类：

- 静态约束：数据库当前状态合不合理
- 动态约束：数据库状态变化的时候是否合理，涉及新旧值比较，比如年龄只能增长

（二）六类完整性约束

上面的两个分类排列组合出六类：

```
静态列级约束
静态元组约束
静态关系约束
动态列级约束
动态元组约束
动态关系约束
```

区分：

先问：限制的是列、同一行多个列、还是多行多表?
再问：是否涉及修改前后的新旧值?

题目描述	类型	常用实现
年龄 14 到 29	静态列级约束	CHECK
性别只能为男/女	静态列级约束	CHECK
学号不能为空	静态列级约束 / 实体完整性的一部分	NOT NULL / PRIMARY KEY
发货量 <= 订货量	静态元组约束	CHECK
学号唯一且不能为空	静态关系约束 / 实体完整性	PRIMARY KEY
学生所在系必须存在	静态关系约束 / 参照完整性	FOREIGN KEY
班级人数不超过 30 人	静态关系约束 / 统计约束	触发器
工资不能低于原工资	动态列级约束	触发器
工资调整不得低于原工资 + 工龄 × 1.5	动态元组约束	触发器
事务一致性、原子性	动态关系约束	事务机制

（三）几类完整性约束条件

域完整性、实体完整性、参照完整性、用户自定义完整性

（1）域完整性

就是**属性的域**，域完整性限制某个属性的取值范围、类型、格式。

常用：

```
数据类型
NOT NULL
CHECK
DEFAULT
```

（2）实体完整性

主键取值唯一，且不能为空，常用PRIMARY KEY。

违反实体完整性时，一般直接报错**拒绝执行**。

（3）参照完整性

外键约束，外键引用的内容必须在对应表中存在，允许为空。

（4）用户自定义完整性

不是上面几个，常用CHECK、触发器

（四）完整性规则五元组

一个规则可以表示为：`(D，O，A，C，P)`

符号	含义	做题时怎么找
D	Data，约束作用的数据对象	哪个表、哪个列、哪个元组
O	Operation，触发检查的操作	INSERT / UPDATE / DELETE 等
A	Assertion，必须满足的断言	真正的约束条件
C	Condition，A 作用的条件	对哪些记录生效
P	Procedure，违反时的处理过程	拒绝、级联、置空、报错等

（五）违约反应

（1）实体完整性：直接拒绝执行

（2）参照完整性：可以拒绝，也可以接受并执行附加动作

- 删除被参照关系中的元组：
    - **级联删除：** 把引用外键的所有记录也一起删除
    - **受限删除：** 若还有引用，则不允许删除
    - **置空值删除：** 允许删除，但是把所有引用处置NULL
- 在参照关系中插入元组，但被参照关系不存在对应元组：
    - **受限插入：** 拒绝插入
    - **递归插入：** 先在被参照关系插入新的，再在参照关系插入
- 修改悲惨找关系的主键：
    - **级联修改**：表中的外键也跟着改。

- **受限修改**：不允许修改主键。
- **置空值修改**：对应外键置空。

（六）Check约束

用于实现静态列级约束和静态元组约束，返回TRUE/FALSE。

用在定义里面，例如：

```
CHECK (salary >= 15000 AND salary <= 100000)
```

表达式可以包含：

```
列名
比较运算符：> < = >= <= <>
逻辑运算符：AND OR NOT
条件谓词：IN LIKE
通配符：_ %
选择符：[0-9] 等
```

不足：`CHECK` 很适合当前行内部的静态约束，但**不适合复杂跨表、跨行、涉及旧值的新旧比较。**

（七）触发器

触发器是实现复杂用户自定义约束的工具，是一种特殊的存储过程。在数据更新的时候自动调用。

对比点	普通存储过程	触发器
调用方式	用户显式调用	数据库自动调用
是否依附对象	可以相对独立	必须依附表、视图、数据库等
参数返回值	可以有参数和返回值	通常无参数、无返回值
触发原因	用户调用	INSERT / UPDATE / DELETE 等事件触发
用途	封装业务逻辑	自动维护约束、日志、级联操作等

（1）E-C-A模型

```
Event - Condition - Action
事件 - 条件 - 动作
```

事件：触发器因为什么被触发，INSERT / UPDATE / DELETE 等

条件：判断是否需要执行动作，WHEN

动作：触发后执行的SQL语句，多条语句要用BEGIN...END

（2）基本语法

```
CREATE [ OR REPLACE ] TRIGGER trigger_name
[BEFORE | AFTER | INSTEAD OF]
trigger_event ON table_reference
[FOR EACH ROW [ WHEN trigger_condition ]]
trigger_body;
```

```
trigger_name：触发器名字
BEFORE / AFTER / INSTEAD OF：触发时机
trigger_event：INSERT / DELETE / UPDATE
ON table_reference：依附在哪个表或视图上
FOR EACH ROW：是否行级触发
WHEN：触发条件
trigger_body：触发动作（多条语句用BEGIN END）
```

（3）触发时机

- BEFORE：原操作执行之前触发
- AFTER：原操作执行之后触发
- INDSTEAD OF：代替原操作（不可更新视图的更新）

（4）分类

- 语句级触发器：默认。每条用户语句触发一次
- 行级触发器：必须写FOR EACH ROW，每影响一行，触发一次

（5）Referencing引用

触发器里经常要用**新值和旧值**

SQL Server 类似有 `Inserted` 表（新插入的行）和 `Deleted` 表（删除的行），OpenGauss、MySQL 可以用 `NEW`、`OLD` 表示新旧值。如：

```
NEW.salary
OLD.salary
```

（6）触发器动作：自定义约束的业务逻辑，可以做更新、插入、拒绝

- 通常用于关联数据更新、维护约束、写日志
- 一般**不能向用户返回查询结果**
- 拒绝操作时要**触发错误或返回空**

eg.

```
CREATE OR REPLACE FUNCTION Stu_num()
RETURNS TRIGGER AS $$
DECLARE
  SSum INT;
BEGIN
  SELECT COUNT(*) INTO SSum
  FROM Student
  WHERE class = NEW.class;
```

```
9
10    IF SSum >= 30 THEN
11      RAISE EXCEPTION 'Too many students in one class';
12      RETURN NULL;
13    END IF;
14
15    RETURN NEW;
16  END
17  $$ LANGUAGE PLPGSQL;
18
19  CREATE TRIGGER TSC
20  BEFORE INSERT ON Student
21  FOR EACH ROW
22  EXECUTE PROCEDURE Stu_num();
```

九、安全

数据库安全包含：安全认证、访问控制、数据保护、数据审计

（一）安全认证

定义：确认试图登录数据库的用户是否被授权访问数据库的过程

登录的时候看用户名密码对不对

认证方式：

- 数据库认证：密码认证
  - 常用
  - 容易被盗，容易被网络攻击，影响性能
- 外部认证
  - 强身份认证：双因素/多因素认证
  - 代理认证：让中央设施对网络中的所有成员进行身份验证。

（二）访问控制

（1）定义：按照用户的身份和权限，控制用户对数据库中数据的访问。

允许/拒绝访问。

关注四件事：

1. **阻止访问**：无权限时，主体不能访问客体
2. **确定访问权限**：判断主体是否有权访问客体
3. **授予访问权限**：给主体访问客体的权限
4. **撤销访问权限**：删除主体对客体的权限

术语	含义	例子
主体	发起访问的人或程序	用户 UserA、角色 RoleA

术语	含义	例子
客体	被访问的数据库对象	表、视图、存储过程、函数

（2）存取权限

存取权限两个要素：数据对象 + 操作类型

权限分类：

- 系统权限：整个系统层面
  - 启动关闭数据库
  - 转储、恢复数据库
  - 创建、删除
- 数据对象权限：对于基本表、视图、存储过程、函数等对象
- 列级权限：比如只允许某些用户看某些特定列
- 行级权限：对行的权限，常用视图
- 连接级权限：用户能不能连接数据库、连接数量、连接方式

（3）DCL数据控制语言

语句	作用
GRANT	授予权限
REVOKE	收回权限

GRANT：

```
GRANT 权限列表
ON 数据对象
TO 用户或角色;
```

```
GRANT SELECT ON TableA TO UserA;
```

REVOKE：

```
REVOKE 权限列表
ON 数据对象
FROM 用户或角色;
```

```
REVOKE SELECT ON TableA FROM UserA;
```

授予列级权限：

```
GRANT SELECT(a, b) ON TableA TO UserA;
```

授予存储过程执行权限：

```
GRANT EXECUTE ON ProcedureA TO RoleA;
```

（4）谁有资格授予或收回权限

1. 数据对象的创建者
2. DBA
3. **拥有传播权限的用户**（**WITH GRANT OPTION**）

第三种，自己有什么，就能给别人转授权什么，例如：

```
GRANT SELECT ON TableA TO UserA WITH GRANT OPTION;
```

（5）数据库角色 Role

定义：角色是**命名的权限集合**，使用角色可以方便地进行授权管理。

```
用户 —— 角色 —— 权限
```

分类：

- 服务器角色：系统内建，系统级管理权限
- 数据库角色：可自建`CREATE ROLE roleA`
- Public角色：默认所有用户都有的权限

角色可以从属于其他角色，即可以**继承**

语句：

创建角色：

```
CREATE ROLE roleA;
```

给角色授权：

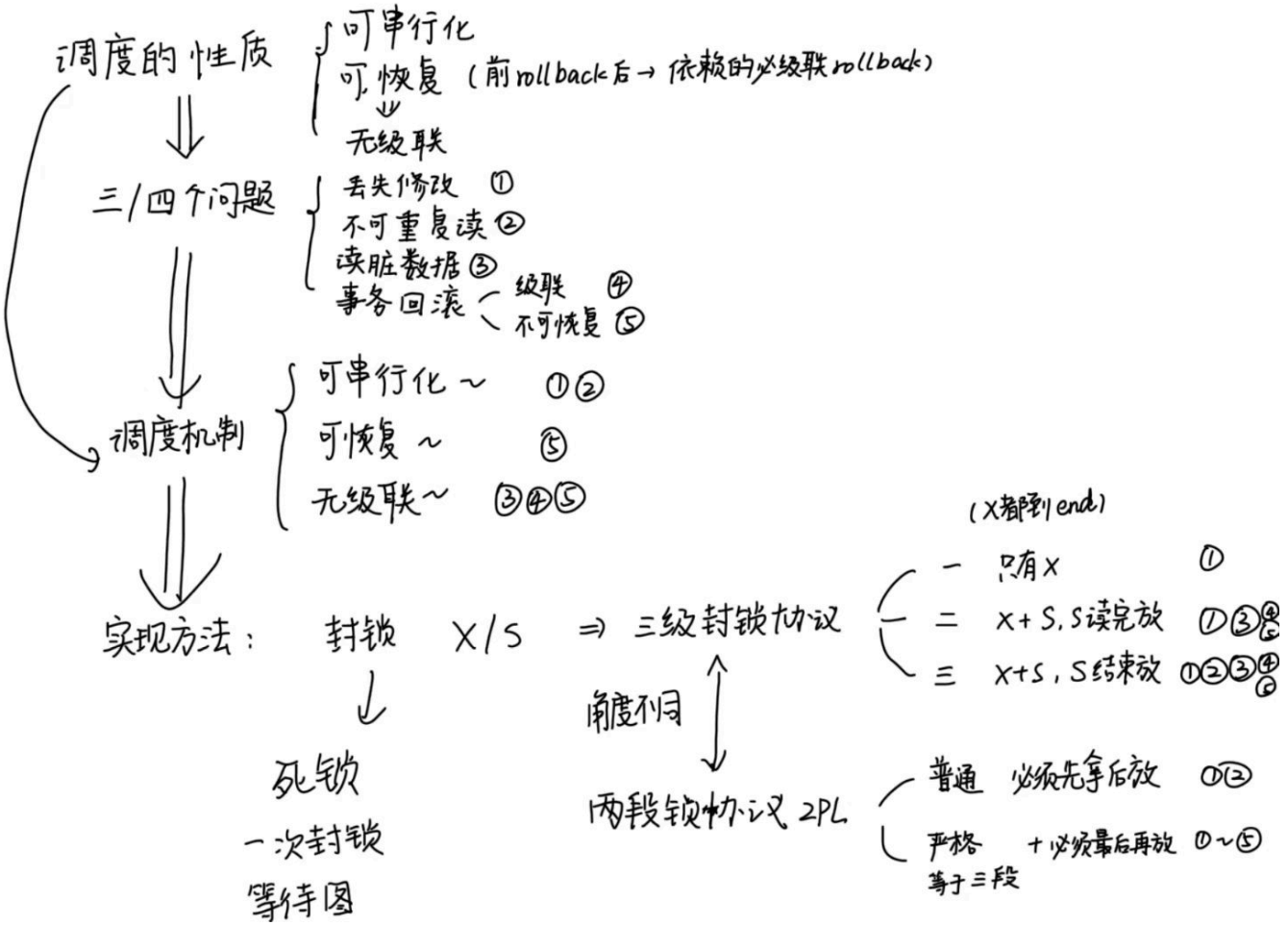
```
GRANT INSERT, DELETE ON TableA TO roleA;
```

把角色赋给用户：

```
GRANT roleA TO UserA;
```

！用户也可以用角色以外的其他权限

十、事务管理



数据库特点：数据资源共享，大部分都是并行处理数据，影响完整性。

数据库的一致性： 在任何时刻用户面对的数据库都是符合现实世界语义逻辑的。并发控制以事务为基本单位，通过对事务操作的数据施加封锁来实现一致性（**动态关系约束**）

（一）事务的定义

事务是用户定义的一个数据库操作序列，要么全做要么不做，不可分割。

- 一个程序可能包含多个事务。
- 事务是**并发控制的基本单位**，是遇到数据库错误后**数据恢复**的处理单位

1. SQL事务的定义方法

如果没有显式定义事务，则DBMS自动划分事务，一般一条DML为一个。

事务开始：

```
BEGIN TRANSACTION;
```

事务正常结束（提交事务，把事务更新结果持久保存）：

```
COMMIT;
```

事务异常结束（回滚事务，撤销已完成更新，恢复到事务开始前）：

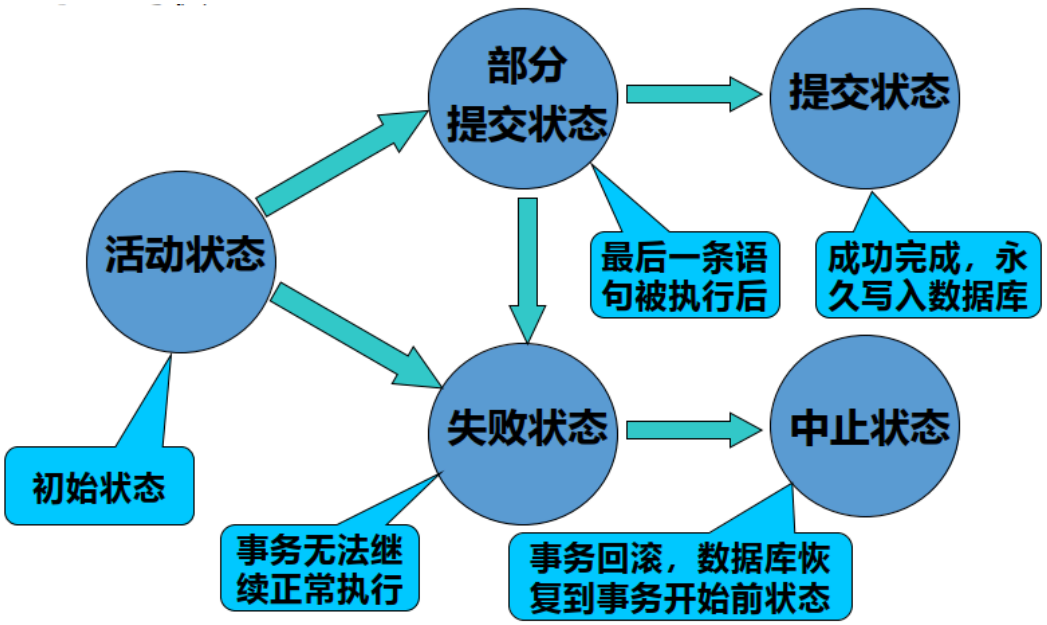
```
ROLLBACK;
```

2. ACID四大特性

字母	名称	核心意思
A	Atomicity 原子性	一个事务要么全做，要么全不做
C	Consistency 一致性	事务前后数据库都满足业务约束，主要看事务结束后规则是否还成立
I	Isolation 隔离性	并发事务之间互不干扰，数据隔离，别的事务不能看其他事务中间状态的数据
D	Durability 持久性	提交后结果永久保存，即使系统故障也无法消失

3. 事务的执行状态

状态	含义
活动状态	初始状态，事务正在执行
部分提交状态	最后一条语句执行完，部分更新，但还没真正永久提交
失败状态	出现错误，事务不能继续执行，下一步就是终止
中止状态	已经回滚完，数据库恢复到事务开始前
提交状态	已经成功完成，结果永久写入数据库



4. 数据库恢复机制

数据库恢复机制可以解决**原子性**、**持久性**，无法解决并发执行下的数据一致性、隔离性。

一种恢复机制：**影子数据库技术**，假设只有一个事务活动，数据更新放影子数据库上，成功则修改数据库指针，错误则删除影子数据库。

（二）并发执行与调度

1. 并发带来的三种数据不一致

（1）**丢失修改**

写写冲突，两个事务同时做出修改，后面的覆盖了前面写入的结果，导致修改丢失

（2）**不可重复读**

读后写后写冲突。一个事务对同一个数据查两次，另一个事务在中间修改。

T1 读取某数据后，T2 **修改、删除或插入**相关数据，使得 T1 再次读取时结果和第一次不同。

其中删除与插入的不可重复读，也称为**幻影现象。**

（3）**读脏数据**

写后读冲突，T2 读取了 T1 尚未提交的数据，而 T1 后来回滚，导致 T2 读到的数据其实不存在。

2. 调度

定义：并发事务中各条指令的实际执行顺序。

- 调度可以交错不同事务的语句，但必须保持同一事务内部原有顺序。
- 串行调度一定正确，同时只有一个事务运行，虽然可能不同顺序结果不一样，但是不会让数据库产生不一致

3. 并行调度可串行化

（1）**并行调度的可串行化**：

- 若每个事务自身都能保证一致性，则事务串行执行也能保证一致性
- **一个并发调度是正确的，当且仅当它的结果等价于某个串行调度。**
- ⬆️判断并行事务正确性的**唯一准则**
- 分为**冲突可串行化**和视图可串行化

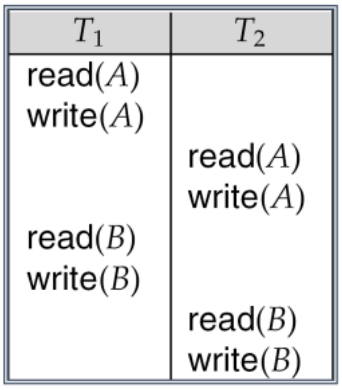
（2）冲突可串行化

冲突：两个指令执行顺序上有要求。条件：

- 来自不同事务
- 访问**同一个数据项**
- 至少一个是写操作（只有读后读没事）

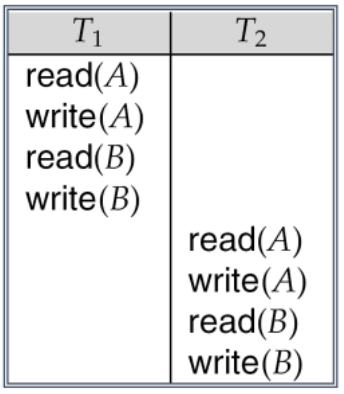
冲突等价： 如果调度 S 可以通过交换一些“不冲突的相邻操作”，变成调度 S'，S 和 S' 冲突等价。

冲突可串行化： 如果 S 能等价变成某个串行调度，那么 S 是冲突可串行化。例如：



T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Schedule 5



T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Schedule 6

4. 前趋图 测试方法

一种冲突可串行化的测试方法。

1. 顶点画不同的事务
2. 找任意两个事务间的冲突操作
3. 如果A事务先访问冲突资源，B后访问，则画一个A到B的箭头，箭头上可以标注资源名称
4. 判断有没有环（跟资源名称没关系）
    - 有：不可冲突可串行化
    - 无：冲突可串行化
5. 若无环，转换为串行调度，使用**拓扑排序**

5. 可恢复调度、无级联调度

可串行化**不能解决**：如果事务回滚，会不会影响已提交的事务读取了错误数据

可恢复调度：如果事务 Tj 读取了事务 Ti 写入的数据，那么 Ti 的**提交**必须发生在 Tj 的提交之前。

```
Tj 读了 Ti 写的数据
=> commit(Ti) 必须在 commit(Tj) 前
```

无级联调度： 如果 Tj 要读 Ti 写的数据，**Ti 必须先提交，Tj 才能读**。(要求更强，避免大量回滚浪费资源)

```
Tj 读 Ti 写的数据
=> commit(Ti) 必须在 read(Tj) 前
```

- 无级联调度一定是可恢复调度
- 可恢复调度不一定是无级联调度

（三）封锁方法：确保调度正确性

1. 封锁基本概念

事务在访问数据对象之前，先申请锁。

- 如果加锁成功，事务才能访问。
- 在锁释放之前，其他事务的访问会被限制。

> 锁并不是从数据底层结构上进行封锁，而是事务必须遵守锁的规则才能生效，**必须保证事务在访问数据之前必须持有锁**。

两种类型：

- **排它锁：** X锁，写锁
  - 如果事务 T 对数据 A 加了 X 锁：T 可以读 A、写 A，其他事务不能再对 A 加任何锁
- **共享锁：** S锁，读锁
  - 如果事务 T 对数据 A 加了 S 锁：T 可以读 A，其他事务可以继续加 S 锁读 A，但不能加 X 锁修改 A

封锁协议： 利用锁来设计一组规则，安排一组并发事务，使调度是**可串行、可恢复的**

- 三级封锁协议
- 两阶段封锁协议

2. 三级封锁协议

设定封锁时机的规则，何时申请、释放锁

（1）一级封锁协议

事务 T 在修改数据 R 之前，必须先对 R 加 X 锁，直到事务结束才释放。（读不需要申请锁，随便读）

- 能解决：丢失修改
- 不能解决：可重复读、不读脏数据

（2）二级封锁协议

写前加 X 锁，X 锁到事务结束；读前加 S 锁，**读完就释放。**

- 能解决：丢失修改，不读脏数据
- 不能解决：可重复读

（3）三级封锁协议

写前加 X 锁，X 锁到事务结束；读前加 S 锁，S 锁也到**事务结束。**

能解决所有问题。

3. 数据库事务隔离级别

数据库对于不同事务的隔离级别。越靠上并发高，越靠下数据更安全

隔离级别	丢失修改	脏读	不可重复读	幻影读
读未提交（另一个事务未提交的数据可以读）	不允许	允许	允许	允许
读已提交（必须已经提交事务才可以读）	不允许	不允许	允许	允许
可重复读（其他事务修改的数据不会对正在进行的事务产生影响）	不允许	不允许	不允许	允许
串行化（一个事务正在读的数据会被完全锁定）	不允许	不允许	不允许	不允许

3. 两阶段封锁协议 2PL

两阶段封锁协议是保证**可串行化**的核心协议。

1. 读 / 写任何数据之前，必须先获得相应锁。
2. 一旦释放了某个锁，以后不能再申请任何新锁。

两个阶段：（必须按顺序）

- 获得封锁，扩展阶段
- 释放封锁，收缩阶段

如果所有并发执行的事务遵守2PL，所有调度是可以串行化的。

- 2PL 调度等价的串行顺序，可以按事务锁定点（一个事务获得最后一个锁的时刻）的先后顺序确定

严格2PL： 在 2PL 基础上，事务获得的锁只有在事务结束时才释放。

- 所有锁都持有到 COMMIT 或 ROLLBACK
- 其他事务不会读到未提交数据，可以避免级联回滚。
- **第三级封锁协议与严格两阶段锁协议一致**（都是到事务结束才释放）

关系图：

```
第三级封锁协议
        ||
严格两阶段封锁协议 Strict 2PL
        ↓
普通两阶段封锁协议 2PL
        ↓
保证冲突可串行化
```

4. 死锁

两段锁协议不能解决死锁。即互相持有资源并互相等待。

- 预防死锁：**一次封锁法**，一次把所有需要的数据都加锁，但是降低并发度
- 死锁诊断与解除：**等待图法**
    - 结点：事务；边：等待关系

- 图中有环则发生死锁
- 解除：选择一个事务回滚

十一、备份与恢复

（一）备份恢复定义

（1）数据库恢复主要保证：

- 原子性 A：未提交事务造成的影响需要UNDO掉
- 持久性 D：事务提交后如果还没有写到磁盘，需要REDO掉

（2）备份恢复分类：

- **逻辑备份恢复：** 用SQL从数据库把数据库导出来，导出的是表结构、数据、SQL 语句或逻辑数据
  - 适合小型迁移，对性能影响大
- **物理备份恢复：** 直接备份数据文件
  - 策略复杂，对数据库影响小

（二）三类故障

故障类型	影响范围	典型原因	恢复核心
事务故障	只影响某个事务本身	输入错误、溢出、完整性约束违反、程序错误、死锁	UNDO 未提交事务
系统故障	影响正在运行的事务，内存缓冲区丢失，外存未损坏	停电、OS/DBMS 错误、CPU 故障	UNDO 未完成事务，REDO 已提交事务
介质故障	磁盘/外存数据丢失，破坏性最大	磁盘损坏、磁头碰撞、强磁场干扰	装入备份副本，再 REDO 成功事务

（三）恢复实现技术

基本原理：**冗余。**需要其他地方多存点数据。

包括两类：**数据转储、日志文件。**

1. 数据转储

数据库在另外的存储设备上备份副本。

（1）**静态转储**

静态转储是在系统中没有运行事务时做备份。特点：

- 转储开始时数据库处于一致状态；
- 转储期间不允许任何存取和修改；
- 优点：实现简单；
- 缺点：可用性差，用户事务要等。

（2）动态转储

动态转储是转储和用户事务并发进行。特点：

- 转储期间允许存取、修改；
- 可用性好；
- 但备份副本本身可能不是一致状态。

所以动态转储恢复必须靠：**后备副本 + 日志文件。**

（3）全量转储 vs 增量转储

类型	做什么	优点	缺点
全量转储	每次备份全部数据文件	恢复方便	备份影响大
增量转储	只备份上次转储后更新过的数据	备份影响小	恢复复杂

2. 登记日志文件

记录事务对数据库的更新操作。

- 不同事务的日志文件交错存储
- **顺序写入**，写入效率高
- 事务操作先写入日志，写成功后再写入数据缓冲区之后再写入数据文件

（1）文件内容

事务开始、结束标志。

事务的数据修改记录

事务有关的内部操作（检查点等）

（2）事务原语

原语	含义	位置变化
`INPUT(X)`	把数据项 X 从磁盘读到内存缓冲区	磁盘 → 缓冲区
`READ(X,t)`	把 X 从缓冲区读到事务本地变量 t	缓冲区 → 本地变量
`WRITE(X,t)`	把本地变量 t 写回缓冲区中的 X	本地变量 → 缓冲区
`OUTPUT(X)`	把缓冲区中的 X 写回磁盘	缓冲区 → 磁盘

（3）缓冲区处理策略

- **Force**：（必须commit直接写磁盘）事务提交时，修改过的数据最晚必须写入磁盘。
  所以已提交事务不会丢，不需要 REDO。
- **No Force**：（不必须，可以先留在内存里）事务提交后，修改可以暂时留在内存，过一会儿再写磁盘。
  所以系统崩溃后，已提交事务可能没写入磁盘，需要 REDO。
- **No Steal**：（不允许提前写）不允许未提交事务的数据写入磁盘。

所以未提交事务崩溃后在内存里自然消失，不需要 UNDO。

- **Steal**：（可以提前写）允许未提交事务的数据提前写入磁盘。
  所以崩溃后磁盘里可能有未提交数据，需要 UNDO。

3. Undo日志

Undo日志记录：

- `<START T>`：事务开始；
- `<COMMIT T>`：事务提交；
- `<ABORT T>`：事务回滚；
- `<T, X, v>`：表示事务 T 修改了数据项 X，**X 修改前的旧值是 v。**

（1）规则

日志和数据先后写**磁盘**的问题：

- 更新操作：日志先写，数据文件后写。
- Commit 操作：数据文件先写，Commit 日志后写。

（2）恢复方法

系统故障后，做法是：

1. 看哪些事务已经完成；
2. 对没有 `<COMMIT T>` 或 `<ABORT T>` 的事务进行 UNDO；
3. 从日志**尾部**往前扫描；
4. 遇到未完成事务的 `<T,X,v>`，就把 X 恢复为旧值 v。

（3）优缺点

优点：保证原子性，撤销未提交事务

缺点：**提交前必须把数据写到磁盘，性能差**

4. Redo日志

Redo日志记录：

- `<START T>`：事务开始；
- `<COMMIT T>`：事务提交；
- `<ABORT T>`：事务回滚；
- `<T, X, v>`：表示事务 T 修改了数据项 X，**X 修改前的新值是 v。**

（1）规则

如果 T 更新了 X，则 `<T,X,v>` 和 `<COMMIT T>` 必须在 X 写入数据文件之前写入日志文件。（这些日志都要先写）

（2）恢复

系统故障后：

1. 找出已经完成的事务；

2. 从日志**开头**正向扫描；（按照时间顺序）
3. 对每个已提交事务做 REDO；
4. 未提交事务忽略。

（3）限制

Redo日志不允许Steal模式，因为日志必须在数据写入之前写入文件。

能解决持久性，但是解决不了Steal带来的原子性问题。

5. Undo/Redo日志

面向No force + Steal模式

记录的是：`<T，X，u，v>`：u为旧值，v为新值

（1）规则

如果事务 T 更新了 X，则 `<T,X,u,v>` 必须在 X 写入数据文件之前写入日志文件。
Commit 记录在数据写入前后都可以。

（2）恢复

恢复步骤是（相当于前面两个结合）：

1. 确定哪些事务完成，哪些事务未完成；
2. 从日志开头正向扫描，对已完成事务 REDO；
3. 从日志末尾反向扫描，对未提交事务 UNDO。

6. 检查点

（1）是什么

检查点记录包含：

1. 建立检查点时刻**所有正在执行的事务清单**；
2. 这些事务**最近一个日志记录**的地址。

还有一个**“重新开始文件”，记录检查点记录在日志文件中的地址。**

（2）建立检查点

1. 把当前日志缓冲区中的所有日志记录写入磁盘日志文件；
2. 在日志文件中写入检查点记录；
3. 把当前数据缓冲区的所有数据记录写入磁盘数据库；
4. 把检查点记录地址写入重新开始文件。

（3）检查点恢复

1. 从重新开始文件找到最后一个检查点地址；
2. 找到最后一个检查点记录；
3. 得到检查点时正在执行的事务；
4. 从检查点开始正向扫描到日志结束，建立已提交和未提交队列；

5. 已提交事务 REDO，未提交事务 UNDO

> 检查点时仍在运行的事务，即使它的某些日志在检查点之前，也可能还要往前找，因为要 UNDO 它的早期操作。

7. ARIES算法

核心流程：

1. **Analysis 分析阶段**
   利用 Checkpoint 和日志找出故障时的活跃事务列表。
2. **Redo 阶段**
   通过 Redo log 恢复数据。注意，ARIES 中未提交事务也可能先 REDO，因为要让数据库回到崩溃瞬间的状态，避免索引、数据页等结构不一致。
3. **Undo 阶段**
   对仍未提交的事务进行回滚，保证原子性。

与普通的区别：REDO的时候先到崩溃时刻，再UNDO未提交事务。

十二、规范化理论

方法：

依赖（两端所有都在Ui中）

小化依赖集）

自然连接之后和原来的是否相等

方面）

依赖，推出来任何一张表的剩余属性，就是无损连接

① 闭包计算：瞪眼法

② 最小依赖集（三条）

③ 投影（全部包含+②）

④ 无损：(1) 2个：定理

$R_1 \cap R_2 \to R_1 - R_2$

或 $R_1 \cap R_2 \to R_2 - R_1$

(2) 列表

列：每个属性

行：关系

有aj 无bij

看依赖 X → Y

↑ ↑

相同 相同

⑤ 依赖

$R(U, F) \Rightarrow R_1 \dots R_n$

$F_1 \cup \dots \cup F_n = G$

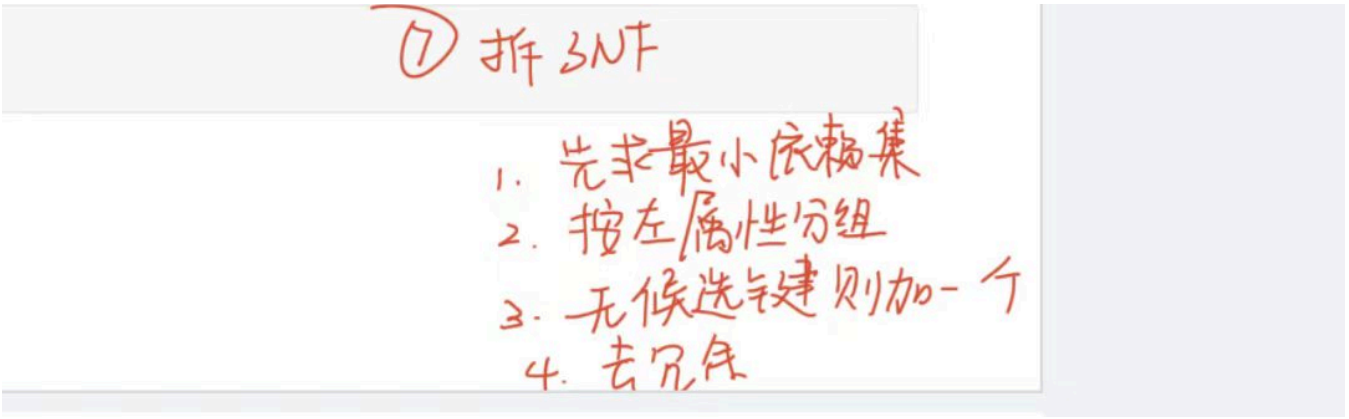
F中每个 A→B

若 A →(G) B 也成立 ✓

⑥ 拆 BCNF

不满足的 X → Y

把 (X, Y), (X, Z)



⑧ 候选键：分L、R、LR、N，L与N必须在候选键，然后再看LR一个一个试，看哪个能推出全集

关系数据库的规范化理论包括三个方面：

- 函数依赖
- 范式 NF
- 模式分解

关系模式可能产生的四类问题：

- 数据冗余
- 插入异常（只有部分数据，没地方可插入）
- 删除异常（删除的同时删掉了其他有用信息）
- 更新异常（一个数据要更新很多个地方）

（一）函数依赖

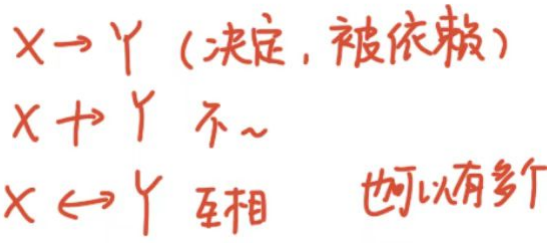
（1）定义

函数依赖：**属性之间的一种逻辑依赖关系**

X、Y是属性集合，X确定就能唯一确定Y，则X决定Y / Y依赖于X

- 函数依赖不能只看当前数据（与时间无关），而是**语义完整性约束**

（2）表示



（3）平凡函数依赖 与 非平凡函数依赖

如果右边是左边的一部分（子集），显然右边依赖于左边，称为**平凡函数依赖。**

其余为非平凡

（4）**投影分解**

对于R（X, Y, Z），X,Y,Z为不相交的属性集合，如果X->Y，则

$$R（X,Y,Z） = R1[X,Y]自然连接R2[X,Z]$$

分成三部分，其中一部分依赖于另一部分，这两个部分单独放一起，被依赖的和剩下的放一起（**因为由X能推出Y，所以有没有Y都一样**）。

（5）分类

完全函数依赖： Y 依赖于整个 X，X为最小的依赖，不可分解（**f**ull）

```
X  -f→  Y
```

部分函数依赖： Y只依赖于X的其中一部分，即X可分解。（**p**art of）

```
X -p→ Y
```

传递函数依赖： X->Y（且不能Y->X，相当于两个一样），Y->Z，所以X->Z。间接决定（**t**ran...）

```
X -t→ Y
```

（二）几种键、码定义

R为关系模式，K为R的一个属性（集），U为R全集

候选键：K为U的完全依赖，就是这个是不可拆分的能确定一个行的属性集，可以有多个

主键： 候选键中选一个

超键： 所有能确定一个行的属性集，范围最大的

主属性： 在任意候选键中出现过的属性

非主属性： 主属性之外的属性

外键： 这个属性是另外一个关系模式的候选键

（三）范式

范式 Normal Form = NF

是**数据库规范化的标准。**

```
1NF ⊇ 2NF ⊇ 3NF ⊇ BCNF ⊇ 4NF ⊇ 5NF
```

分析步骤：要先分析候选键有哪些

1. 第一范式 1NF

关系模式的所有属性都是简单属性（只有一个值），不存在多值/组合。

2. 第二范式 2NF

1NF + **每个非主属性都完全函数依赖于每个候选键**

（1）理解

也就是每次选一个候选键，显然他能唯一确定任意一个非主属性，所以非主属性一定依赖于每一个候选键。

-> 要求不能是**部分依赖**，必须是完全依赖

解决了部分依赖问题，没解决传递依赖

（2）规范化

投影分解。“一事一表”

> 把单独的事情拆出来，即原本部分依赖的关系，把完全依赖的部分拆出来，剩下的是另外一部分

对于R（X,Y,Z），X是主属性，Y、Z是非主属性。本来Y部分依赖于X，把X拆成两部分X1，X2

则X1、Y组成新的，X、Z（X1+ （X2+Z））是另外一个。

（3）特点

- 所有候选键都只有一个属性，必然是2NF +
- 所有属性都是主属性，必然是2NF +

3. 第三范式 3NF

2NF + **每个非主属性都不传递依赖于 R 的每个候选键**

不再有传递关系。

规范化：还是一样，把一个关系只描述一个实体或实体间的联系。

4. BC范式 BCNF

1NF + **对于每个非平凡函数依赖 X → Y，X 都必须包含一个候选键**

解决的是主属性部分依赖于键的问题。

（四）非规范化设计

场景：

```
大量频繁查询需要多表连接
主要应用总是把几个表连接起来查
查询需要复杂计算、临时表、聚合统计
```

常见技术：

- 增加冗余列
- 增加派生列（减少计算、集函数使用）
- 重新组表（经常连接的两个表组成一个表）
- 水平分割（行分割）
- 垂直分割（列分割，常用查询更快）

（五）模式分解

讲的是如何进行投影分解，一般有很多不同的结果。

目标：分解后的关系模式与原关系模式等价。

模式分解可能遇到的问题：

- 是无损连接（自然连接之后拼出来的没区别），但是丢失了一部分的函数依赖
- 不仅丢失了函数依赖，还是有损的

1. 形式化定义

对于一个模式，其可以表示为`R<U, F>`，U为属性全集，F为函数依赖集。

分解可以写作：

```
ρ = { R1<U1, F1>, R2<U2, F2>, ..., Rn<Un, Fn> }
```

要求：`U = U1 ∪ U2 ∪ ... ∪ Un`，且不存在`Ui ⊆ Uj`

（意思是，几个属性集合拼起来能凑成全集，且不存在包含关系）

F在Ui上投影：原来的函数依赖集 F 中，哪些依赖可以只用 Ui 里的属性表达出来，这些依赖就是 F 在 Ui 上的投影。

形式化表达：

```
πUi(F) = { X → Y | X → Y ∈ F+ 且 X、Y 都属于 Ui }
```

2. Armstrong公理系统

⚠️ **AB = {A, B}**，所以不用看所有排列组合，只用管元素

（1）逻辑蕴含

> 从F这些函数依赖集中，可以推导出别的依赖

如果所有满足 F 的关系，都一定满足：

```
X → Y
```

那么说：

```
F 逻辑蕴含 X → Y
```

记作大概就是：

```
F ⊨ X → Y
```

（2）三条基本公理

- **自反律：** `若 Y ⊆ X，则 X → Y`（Y是X的一部分，平凡依赖）
- **增广律**：`若 X → Y，则 XZ → YZ`
- **传递律：** `若 X → Y，Y → Z，则 X → Z`

推导出一些定理：

- 合并规则：`若 X → Y，X → Z，则 X → YZ`
- 分解规则：`若 X → YZ，则 X → Y，X → Z`
- 伪传递规则：`若 X → Y，WY → Z，则 XW → Z`

（3）属性闭包

函数依赖闭包 `F+` 是 F 能推出的全部依赖。

X 关于 F 的**属性闭包**：X_F+或者X+，`X+` **是由 X （一个/一组属性）出发，根据 F（所有依赖）能推出的所有属性集合。**

用途：

- 如果一组属性Y属于X+，则X->Y成立
- 也可以用于算候选键，如果X+ = U，则X为超键，然后再看子集

闭包计算方法：

给定：

```
U = 属性全集
F = 函数依赖集
求 X+
```

做法：

```
1. 初始 X+ = X
2. 扫描 F 中所有函数依赖
3. 如果某个依赖左边已经包含在 X+ 里，就把右边加入 X+
4. 反复扫描，直到 X+ 不再变化
```

（4）公理系统的有效性和完备性

有效性：这个公理推出的函数依赖一定是真的

完备性：所有真的函数依赖都能被公理推出来

（5）函数依赖集等价

两个依赖集 F 和 G 如果推出能力完全一样，即：

```
F+ = G+
```

就说它们**等价**，或者互相覆盖。**（这里说的是函数依赖，不是属性）**

判断方法：

```
F+ = G+
等价于：
F ⊆ G+
并且
G ⊆ F+
```

实际做题时：

> 要判断 F 中某个 `X → Y` 是否能由 G 推出，就算 `X` 关于 G 的闭包，看 Y 是否在里面。

（6）最小依赖集

函数依赖集（最小覆盖）满足以下三个条件

1. 每个属性右边只有一个属性（不能A->BC）
2. 不能有多余的函数依赖（即存在函数依赖可以被其他推出）
3. 左边不能有多余属性（即存在AB->C的时候，不能同时满足A->C）

求最小依赖集方法：

- 检查每个依赖，右边不能有多余一个属性，多的拆开
- 拆掉多余依赖：检查每个依赖，尝试先把它删掉，如果删掉之后右边的部分（X->A中的A），从X开始推，还能被推出来，则这个依赖多余了
- 删左边多余属性：检查每一个依赖，检查依赖左边每一个属性，尝试去掉一个，从剪掉的左边开始推，如果还能推出来右边的结果，则删掉

最小依赖集不一定唯一。

3. 模式分解正确性

计算F在Ui上的投影（即依赖函数集）方法：

1. 只考虑 Ui 中的属性
2. 找 Ui 内部能由原 F 推出的函数依赖（两端所有都在Ui中）
3. **去掉冗余依赖**，得到投影（最小化依赖集）

（1）分解正确性判断：

- 是否是无损连接（数据方面）：自然连接之后和原来的是否相等
- 分解是否保持函数依赖（依赖方面）

二者相互独立。

（2）无损连接性判断一：定理法

如果两关系分解：R 分解为 R1 和 R2

常使用**定理法**：

```
若 R1 ∩ R2 → R1 - R2
或 R1 ∩ R2 → R2 - R1
则分解无损。
```

两个关系的公共属性， 能用R的函数依赖，推出来任何一张表的剩余属性，就是无损连接

（3）无损连接判断二：列表法

分解为多个关系用列表法。

假设原关系：

```
R(A1, A2, ..., An)
```

分解为：

```
R1, R2, ..., Rk
```

构造一个表：

```
行：每个子关系 Ri
列：原关系每个属性 Aj
```

填表规则：

```
如果 Aj 属于 Ri，填 aj
如果 Aj 不属于 Ri，填 bij
```

也就是：

```
属于该子模式的属性填 a
不属于的填 b
```

然后用函数依赖反复改表。

如果有依赖：

```
X → Y
```

并且某两行在 X 对应列相同，那么它们在 Y 对应列也应该相同。

修改时：

```
如果某列中有 aj，就统一改成 aj
如果没有 aj，就统一成某个 bij
```

最后判断：

```
若某一行全变成 a1, a2, ..., an，则无损连接。
否则有损。
```

■ 现有关系模式R<U,F>,U={A,B,C}，F={B→C}

R的一个分解为R_1(A,B), R_2(B,C)，该分解是否无损连接性?

1) 构造初始表格

A	B	C
a1	a2	b13
b11	a2	a3

2) 根据B→C修改表格

A	B	C
a1	a2	a3
b11	a2	a3

第一行符合a1,a2,...,an的情况，判断该分解无损连接性

（4）函数依赖保持性判断

原来的每条依赖，都能不能只靠分解后的各表约束推出?

定义:

如果分解为:

```
ρ = {R1, R2, ..., Rn}
```

每个子模式上的投影依赖为:

```
F1, F2, ..., Fn
```

令:

```
G = F1 ∪ F2 ∪ ... ∪ Fn
```

如果:

```
G 能推出 F 中每一个函数依赖
```

则分解保持函数依赖。

解法:

对原 F 中每个依赖:

```
X → Y
```

计算：

```
X 关于 G 的闭包 X_G+
```

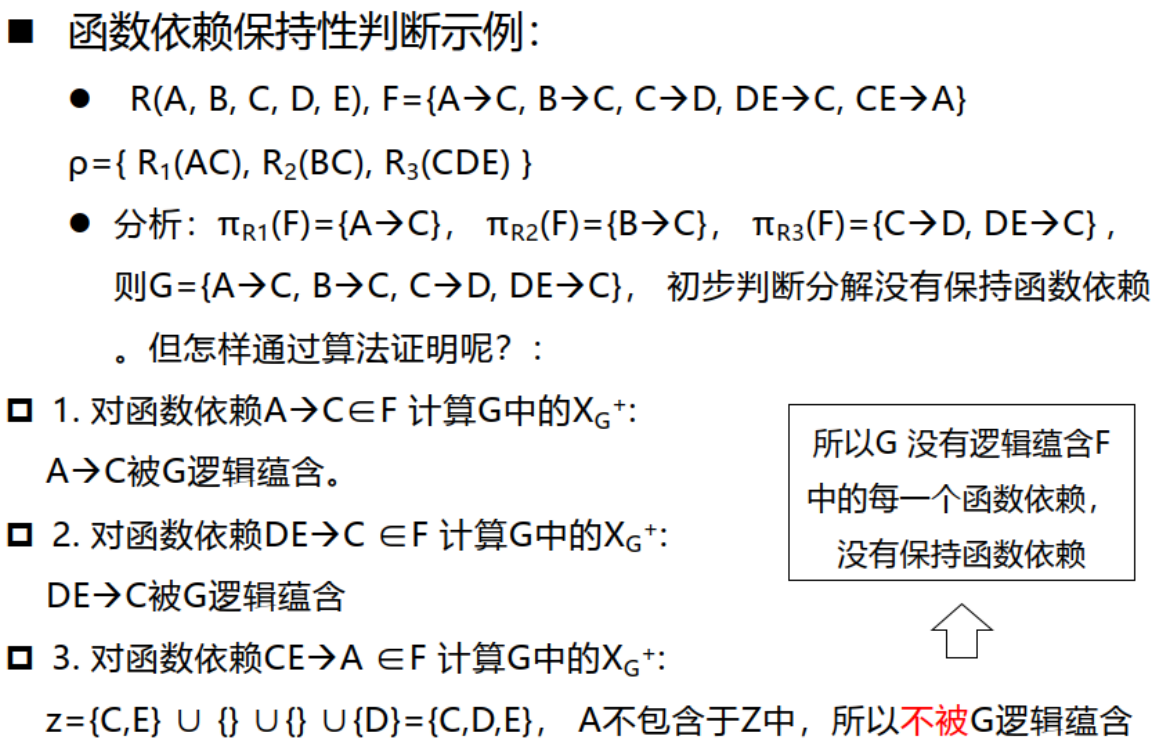
如果：

```
Y ⊆ X_G+
```

说明这条依赖被保留。

所有依赖都能保留，则分解保持函数依赖。

只要有一条不能推出，就不保持



4. 模式分解方法

分解到3NF和BCNF。

（1）BCNF

可以无损连接地分解到 BCNF，但可能丢失部分函数依赖。

从原关系开始：

```
ρ = {R}
```

如果某个关系模式 S 不是 BCNF，则存在某个非平凡依赖：

```
X → A
```

其中：

```
X 不是 S 的超键
A ∉ X
```

这就是违反 BCNF 的依赖。

然后把 S 分解为：

```
S1 = X ∪ A
S2 = S - A
```

也就是：

```
一个表放 X 和 A
另一个表保留 S 中除了 A 以外的属性
```

反复做，直到每个子模式都是 BCNF。

（2）3NF：保持函数依赖的分解法

不能保证无损连接。

第一步：**先求最小依赖集**

给定：

```
R(U, F)
```

先求 F 的最小覆盖：

```
Fm
```

第二步：**根据最小依赖集建表**

对每个依赖：

```
X → A
```

建一个关系：

```
XA
```

如果有多个依赖左边相同：

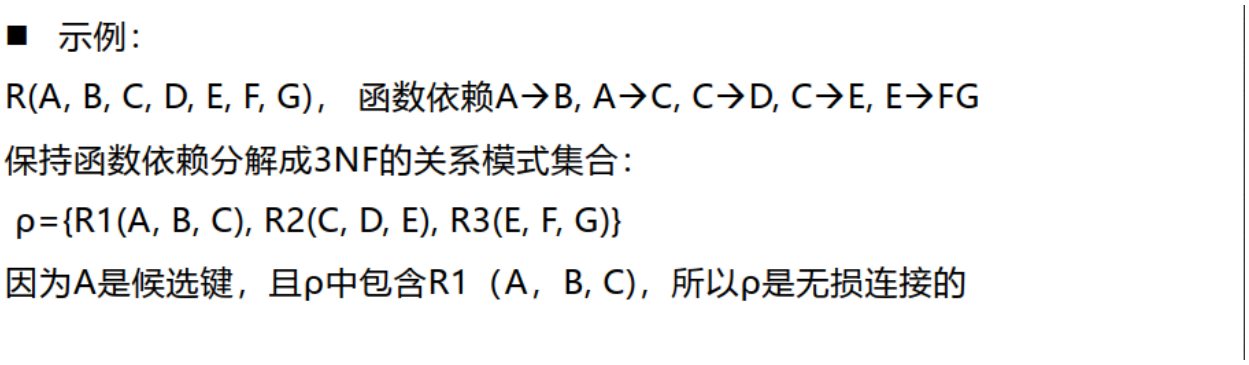
```
X → A1
X → A2
...
X → Am
```

则合并成一个关系：

```
X A1 A2 ... Am
```

也就是**按左部属性分组。**

如果有些属性完全没出现在 F 里，可以单独组成一个关系。



（3）3NF：既保持依赖又无损连接的分解法

如果分解结果中没有任何一个关系包含原关系的候选键，就**把一个候选键单独作为一个关系加进去。**

1. 先按最小依赖集构造保持依赖的 3NF 分解 ρ
2. 求原关系的候选键 X
3. 如果 ρ 中没有任何关系包含 X，就加入关系 X
4. **删除被其他关系包含的冗余关系**

十四、数据库设计

（一）概述

（1）两方面：

- **静态结构设计**
  - 概念、逻辑、物理设计
- **动态行为设计**
  - 数据库应用程序设计

（2）设计方法

- 直观设计、手工试凑
- **新奥尔良法**：将数据库设计分为**需求分析、概念设计、逻辑设计、物理设计**几个阶段

（3）常用设计方法

- 基于 E-R 模型的方法：在需求分析基础上，用 **E-R 图** 构造现实世界中实体及联系，再转换为某个 DBMS 支持的数据模型。
- 基于 3NF 的方法：先确定所有属性和属性间依赖，把它们放到一个大的关系模式里，再根据 3NF 规范化要求分解成若干关系模式。

- 基于视图的方法：先为每个应用建立自己的局部视图，再把各个视图合并成整体概念模式。

（4）数据库设计过程

需求分析 → 概念结构设计 → 逻辑结构设计 → 物理设计 → 数据库实施 → 运行和维护

（二）需求分析

（1）四个要求：信息要求、处理要求（功能）、安全性要求、完整性要求。

（2）分析表达用户需求方法：

- **自顶向下的结构化分析方法 SA （数据流图、数据字典）**
    - 也就是先看整个系统，再拆成子系统，再拆成更小功能。
    - **数据流图 DFD:** 它是从实际系统中抽象出来的，用特定符号反映系统数据传递和变换过程的图。
        - 要画不同层级的图
        - 箭头数据流，圆形为处理，方形为实体
    - **数据字典 DD：** 各类数据描述的集合
        - 数据项、数据结构、数据流、数据存储、处理过程
- 面向对象分析方法

（3）需求分析结果：**需求规格说明书**

需求规约是概念设计的主要依据

（三）概念结构设计

把需求分析得到的用户需求抽象为信息结构，即**概念模型。**

自顶向下地进行需求分析，自底向上地设计概念结构。

（1）步骤：

第一步：抽象数据并设计局部视图
第二步：集成局部视图，得到全局概念结构

（2）**数据抽象**：数据抽象就是从现实的人、物、事、概念中抽取共同特性，忽略不重要细节，用概念模型描述。

常用抽象方法：

- 分类：把同类对象归为实体型，比如学生、教师、课程。
- 聚集：把多个对象或属性组合成更大的整体，比如订单由订单项组成。
- 概括：抽出更一般的上层类型，比如本科生、研究生概括为学生。

（3）局部ER图设计

1. 选择局部应用：看中层数据流图。
2. 逐一设计分 E-R 图：标定局部应用中的实体、属性、码、实体间联系。
3. 集成局部视图：合并 + 修改与重构
    - 冲突：
        - 属性冲突

        - 命名冲突
        - 结构冲突
    - 消除不必要冗余：
        - 冗余数据、冗余联系
        - 使用规范化理论
    - 消除冗余后是**基本E-R图**
    - 数据模型优化（数据依赖理论）

（四）逻辑结构设计

逻辑结构设计的任务：把概念结构转成某个 DBMS 支持的数据模型，并优化。

（1）三项主要内容

- E-R 图向关系模型转换；
- 数据模型优化；
- 设计用户子模式（视图）

（五）关系候选键选择

1. 四类属性

对于关系模式：

```
R<U, F>
```

U 是全部属性集合，F 是函数依赖集合。

根据属性在 F 中出现的位置，分为：

- L 类属性

只在函数依赖左部出现。

结论：
L 类属性必然属于候选键。

- R 类属性

只在函数依赖右部出现。

结论：
R 类属性必然不属于候选键。

- LR 类属性

左部和右部都出现过。

结论：
LR 类属性可能属于候选键。

- N 类属性

既不在左部出现，也不在右部出现。

结论：
N 类属性必然属于候选键。

速记：

```
L、N 必进候选键
R 必不进候选键
LR 可能进
```

求候选键套路：

第一步：找 L 类和 N 类属性，设为 X。

第二步：求 X^+。

如果：

```
X⁺ = U
```

那么 X 是唯一候选键。

如果：

```
X⁺ ≠ U
```

就从 LR 类属性里试着加入一个或多个属性，再求闭包。

只要某个属性组闭包等于 U，并且它不包含更小候选键，它就是候选键。

十五、云数据库

NoSQL数据库

集群上比较强，高并发、高扩展、高可用，海量数据

（1）关系数据库

优势：以完善的关系代数理论作为基础，有严格的标准，支持事务ACID四性，借助索引机制可以实现高效的查询，技术成熟，有专业公司的技术支持

劣势：可扩展性较差，无法较好支持海量数据存储，数据模型过于死板、无法较好支持Web2.0应用，事务机制影响了系统的整体性能等

（2）NoSQL数据库

优势：可以支持超大规模数据存储，灵活的数据模型可以很好地支持Web2.0应用，具有强大的横向扩展能力等

劣势：缺乏数学理论基础，复杂查询性能不高，大都不能实现事务强一致性，很难实现数据完整性，技术尚不成熟，缺乏专业团队的技术支持，维护较困难等

NoSQL技术特点

CAP理论

（A的含义不一样）

- **C（Consistency）：一致性**，是指任何一个读操作总是能够读到之前完成的写操作的结果，也就是在分布式环境中，多点的数据是一致的，或者说，所有节点在同一时间具有相同的数据
- **A（Availability）：可用性，**是指快速获取数据，可以在确定的时间内返回操作结果，保证每个请求不管成功或者失败都有响应；
- **P（Tolerance of Network Partition）：分区容忍性，**是指当出现网络分区的情况时（即系统中的一部分节点无法和其他节点进行通信），分离的系统也能够正常运行，也就是说，系统中任意信息的丢失或失败不会影响系统的继续运作。

CAP不可兼得：三色图

BASE理论

BASE理论

BASE的基本含义是**基本可用**（Basically Availble）、**软状态**（Soft-state）和**最终一致性**（Eventual consistency）：

- **基本可用**
  - 指一个分布式系统的一部分发生问题变得不可用时，其他部分仍然可以正常使用，也就是允许分区失败的情形出现
- **软状态**
  - 是与“硬状态（hard-state）”相对应的一种提法
  - “硬状态” 是数据库保存的数据可以保证数据一直是正确的(一致性)
  - “软状态”是指状态可以有一段时间不同步，具有一定的滞后性

BASE理论

- **最终一致性**
  - **一致性**的类型包括**强一致性**和**弱一致性**
  - 二者的主要区别在于高并发的数据访问操作下，后续操作是否能够获取最新的数据
  - **强一致性**：当执行完一次更新操作后，后续**其他读操作**就**可以保证**读到更新后的最新数据
  - **弱一致性**：如果**不能保证**后续访问读到的都是更新后的最新数据
  - **最终一致性**：是弱一致性的一种特例，允许后续的访问操作可以暂时读不到更新后的数据，但是经过一段时间之后，必须最终读到更新后的数据
  - 最常见的实现最终一致性的系统是DNS（域名系统）。一个域名更新操作根据配置的形式被分发出去，并结合有过期机制的缓存；最终所有的客户端可以看到最新的值。